GENERATION AND OPTIMIZATION OF SPACING FIELDS

By
Max David Collao

Approved:

Steve Karman Jr.
Professor of Computational Engineering
(Director of Thesis)

Daniel Hyams
Associate Professor of Computational Engineering
(Committee Member)

William H. Sutton
Dean of College of Engineering and Computer Science

Chad Burdyshaw
Research Associate Professor of Computational Engineering
(Committee Member)

Jerald Ainsworth
Dean of the Graduate School

GENERATION AND OPTIMIZATION OF SPACING FIELDS

By
Max David Collao

A Thesis
Submitted to the faculty of
The University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computational Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2011

ABSTRACT

Meshes are used to discretize space for computational fluid dynamics (CFD) simulations. Mesh adaptation through refinement and smoothing can improve the accuracy of the CFD solution. In order to perform adaptive refinement or smoothing a spacing field is needed to define the desired edge sizes in the mesh. The objectives of this research are to generate spacing fields from existing CFD solutions and optimize this spacing information for efficient use by programs to perform adaptive refinement or smoothing. All work was done on 2D meshes with the intention of gaining knowledge and experience for later application to 3D meshes. The program written to generate spacing fields is presented. Images depicting spacing fields created from different meshes using this program are shown. Next, a library of functions created to store and retrieve spacing information is presented. Finally, the shortcomings of the developed software as well as ideas for future research are discussed.

DEDICATION

I dedicate this work to Maximo, Gloria, Joel, Jairo, and Elisabet. Because their support is strong and their love feels very warm in spite of the distance.

ACKNOWLEDGEMENTS

I thank God from whom all blessings flow.

I would like to give many thanks to my advisor, Dr. Steve Karman Jr., for sharing his knowledge and for his phrase "there is no stupid question!" Undoubtedly, it has been a very constructive experience to work with him.

Very much appreciation goes to Wally Edmonson, Srijith Rajamohan, Bill Brock, Ashish Gupta, Ethan Hereth, Taylor Erwin, Bruce Hilbert, Cameron Druyor, and Matt O'Connell. Because without their help at some point of this journey I would have not been able to make it to this point.

Last but definitely not least, I want to thank my good friends at the SimCenter and my three peruvian friends in Nashville. Because they became my family in America.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

M, Tensor matrix

R, Rotating matrix

λ, Scaling matrix

$e_{\hat{1},2}$, Principal direction unit vectors

w, Weighting function

$h_{1,2}$, Desired spacing

f, Function

w, Equally distributed weighting function

Q, Item property

f, Scalar function

∇f, Gradient of scalar function

Ω, Volume

S, Surface

$\hat{n}$, Surface unit vector

Af, Adaptive function

δd, Edge length

δd, Spacing for a node

P, Power of the edge length

n, Multiple for standard deviation

df, Change in the scalar function

dx, Change in the x position

f_{right}, Scalar function at right side of region

f_{left}, Scalar function at left side of region

x_{right}, x at right side of region

x_{left}, x at left side of region

CHAPTER 1

INTRODUCTION

One of the most important concepts applied in computational fluid dynamics (CFD) is the discretization of a domain by using a mesh on which the simulation of a flow is to be obtained. In CFD, meshes are composed by a set of nodes or points distributed over the domain's boundaries and the interior. Edges are lines connecting two nodes. For a domain in two dimensions (2D), cells are defined by a set of edges that enclose a space and most commonly have the shapes of triangles and quadrilaterals. For a domain in three dimensions (3D), cells are typically defined by a set of triangular or quadrilateral shaped faces that enclose a space, which are defined similar to the way cells are defined in 2D. Three-dimensional cells are commonly shaped as tetrahedra, pyramids, prisms, and hexahedra. Meshes are classified as structured or unstructured, based on the indexing of the nodes. If an indexing scheme can be implied or inferred, then the mesh type is known as structured [1].

The process of generating optimal meshes for accurate solutions can be very tedious and repetitive. Initially, the person creating the mesh may not have an a priori knowledge of the physical phenomena revealed by the CFD solutions. Using mesh generation software, such as Pointwise [2], he or she will create an initial mesh and determine the spacing between nodes using educated guesses based on previous experience. Once CFD solutions are obtained, they are examined and the locations of the mesh showing results with poor accuracy are identified. Next, the mesh is recreated or modified to improve results in the

areas of the previous mesh where the solutions were not satisfactory. New solutions on the new mesh are obtained, and the process repeats until the CFD solutions are satisfactory.

The process of modifying meshes based on solutions obtained previously is called mesh adaptation. Mesh smoothing can be performed on a mesh in order to smooth the mesh, meaning to improve the quality of cell elements (avoid cell skewness, negative volumes, etc). Mesh adaptation can also be performed to refine the mesh, meaning to add or cluster nodes in regions where the solutions change rapidly. Methods have been developed to automate this process of optimizing meshes based on CFD solution. One such method makes utilization of spacing fields.

This thesis presents the research work done by the author on spacing fields. Efforts were made in order to improve the quality of spacing information in the fields, minimize the amount of information, and speed up the process of retrieving it. All work done was for 2D meshes in order to gain experience and apply it later in 3D. All code was developed using the C++ language, making extensive use of classes. First, the reader is presented with a brief review of how spacing fields have found application in the mesh optimization process. Then, the intermediate calculations necessary to obtain the collection of spacing data across a grid, that make up the spacing field, are explained. The data structure model used for storing and organizing spacing information will be presented as well. Next, the process followed for generating, storing, and retrieving spacing information will be presented, including the algorithms used in the computer programs. Finally, a summary of the work done will be described, and the areas for potentially improving the processes of storing and retrieving spacing information will be highlighted.

CHAPTER 2

BACKGROUND

For over three decades, a great amount of work has been devoted to developing automated methods for mesh adaptation. Among other things, such methods involve solving differential equations. Mesh researchers have been interested in finding the location of nodes in a mesh by solving partial differential equations using numerical techniques [3]. The following sections give insight into some of the schemes developed over the years for solving such equations. The last section provides insight on spacing fields and how they were used in this research.

Adaptive Smoothing and Refinement

Early schemes developed for adaptive meshing made use of the idea of equidistribution. Anderson lists in reference [3] the work done by various researchers on schemes seeking to adjust meshes in regions where solution gradients were large. He identified the use of equidistribution of functions or errors in such schemes, and pointed out some factors that were limiting to their application.

Elliptic smoothing methods have been among the most popular schemes used for adaptive smoothing in the last several years. Commercial mesh generation packages, such as Pointwise and Gridgen, make use of them for mesh optimization [2] [4]. The elliptic smoothing methods are based on the solution of a system of elliptic partial differential equations [3]. Such methods possess smoothness and robustness qualities that are desired for the generation of high quality meshes [5]. The most popular method is based on the

Winslow elliptic smoothing equations. They were proposed by Winslow and were derived from Poisson's equation for a parameter distribution over a region [6].

Structured meshes lend themselves very well to elliptic smoothing because of the implied computational mesh they define. This is not true for unstructured smoothing. Because of this, elliptic smoothing methods were neglected for unstructured meshes for many years. In recent years, researchers have deviced ways to use elliptic smoothing on unstructured meshes without requiring a computational mesh. One of them defines desired edge sizes through Riemannian tensors [7].

Spacing Fields

A spacing field is a collection of items that specify desired spacing at different locations over an entire mesh. The items may specify different spacing in two orthogonal directions, or define a uniform spacing in all directions. The items also specify the region of the grid, called the extent, over which the desired spacing applies.

The spacing information may be defined by a Riemannian metric tensor or by a scalar. If the desired spacing is defined to apply in all directions on its extent, then it is represented by a scalar. However, it may be necessary to define different spacings in two directions. Riemannian metric tensors work very well for this purpose because of their ability to store different spacing information in different directions in a compact manner.

The items in spacing fields are organized in these research through the use of quadtrees. Because, the spacing information is usually defined for nodes or for cells, the items in a spacing field may be numerous and retrieving such information at specific points of the grid can be computationally demanding. In order to speed up the process of retrieving such information, the items are organized in a quadtree where information can be accessed according to their location in the grid and not according to their position in an array.

CHAPTER 3

RIEMANNIAN METRIC TENSORS

In the mesh optimization process, it is often necessary to specify how much change the length of an edge needs and in which direction. Riemannian metric tensors serve this purpose well since they can store this information in a very compact manner, and through manipulation of the tensors they provide a way to decide if the length of an edge is appropriate or not.

Definition

A Riemannian metric tensor is a symmetric positive definite matrix M obtained from the product of a rotation matrix R and a scaling matrix λ [7], as shown in equation (3.1).

$$M = [R][\lambda][R^{-1}] \tag{3.1}$$

Stored in the columns of the rotation matrix are the eigenvectors of M, which represent the principal directions, the directions in which a desired length is specified. They are obtained from the gradients of the scalar function, in this case solution data. The first direction is obtained from normalizing the gradient vector, and the second one is the unit vector normal to the gradient, as shown in figure 3.1.

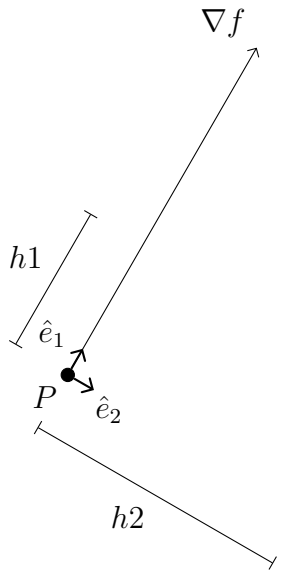

Figure 3.1 Components of spacing information

The scaling matrix λ is a diagonal matrix containing the eigenvalues of M. For adaptation, they are defined as the inverse square of the spacing along each of the principal directions. Equation (3.2) summarizes the explanation above.

$$M = \begin{bmatrix} \vec{e}_1 & \vec{e}_2 \end{bmatrix} \begin{pmatrix} {h_1}^{-2} & 0 \\ 0 & {h_2}^{-2} \end{pmatrix} \begin{bmatrix} \vec{e}_1^T \\ \vec{e}_2^T \end{bmatrix} \tag{3.2}$$

Gradients

The gradients of the solution across the mesh are calculated using a derivation of the theorems of Green and Gauss [8]. The theorems express that the volume integral of the gradient of a scalar function f (over a closed volume defined by a surface) is equivalent to a surface integral of the scalar function, as shown in equation (3.3) [9].

$$\int_\Omega \nabla f \, d\Omega = \int_S f \hat{n} \, dS \tag{3.3}$$

In the equation above the surface normal is represented by $\hat{n}$, and S and Ω stand for surface and volume. From this relation, the equation used for gradient calculation is obtained by approximating the integral over the surface and assuming that the gradient of the function is constant over the control volume, as seen through equations (3.4).

$$\int_{\Omega} \nabla f \, d\Omega = \nabla f \, \Omega = \int_{S} f \hat{n} \, dS$$
$$\nabla f = \frac{\int_S f \hat{n} \, dS}{\Omega} \tag{3.4}$$

Equidistribution

The concept of equidistribution can be better explained by a simple analogy, similar to the one given by Anderson in reference [3], using the relation in equation (3.5) defined for any set of elements.

$$f(w) = wQ \tag{3.5}$$

Suppose Q is any property of each element, and f is a function of w. If f is normalized across all elements w must be adjusted according to the property of each element. In this example, w is called "the weighting function", and it is said to be equidistributed over all the elements.

In the approximation of the current and desired spacings along the principal directions, the same concept of equidistribution is applied. The relation used to calculate the adaptive function is given by equation (3.6).

$$Af = |\nabla f \cdot \hat{e}| \Delta d^P \tag{3.6}$$

Where Δd is the spacing between two nodes at an edge, $\hat{e}$ is the edge vector normalized. The parameter P is defined by the user to provide more influence when adapting the mesh in regions where the grid spacing is larger as opposed to regions of small spacing in the presence of discontinuities such as shocks [7]. In a similar way to the analogy given above, once the adaptive functions have been calculated, a single of value Af is selected for the nodes in the mesh based on calculations of the mean and standard deviation of the functions. Then, equation (3.6) is solved for Δd to obtain equation (3.7). A new value of the spacing at each node along the gradient is calculated after replacing the normalized adaptation function by the local value, and replacing the normalized gradient direction by the unit edge vector.

$$\Delta d_{new} = \sqrt[P]{\frac{Af_{threshold}}{|\nabla f \cdot \hat{e}|}} \tag{3.7}$$

Algorithm for Obtaining the Spacing Field

Calculation of Gradients at Cells

The first step in the calculation of the Riemannian tensors is calculating the gradients at the cells. Every cell in the mesh is visited and equation (3.4) is applied. For each edge of the cell the normal vector to the edge pointing out of the cell, and the average of the scalar function at the edge nodes are calculated. The average scalar functions are multiplied by each of the components of the edge normal vectors. Finally, the summation of such product for each component is divided by the area of the cell. These operations can be summarized by equation (3.8) for a quadrilateral cell represented in figure 3.2.

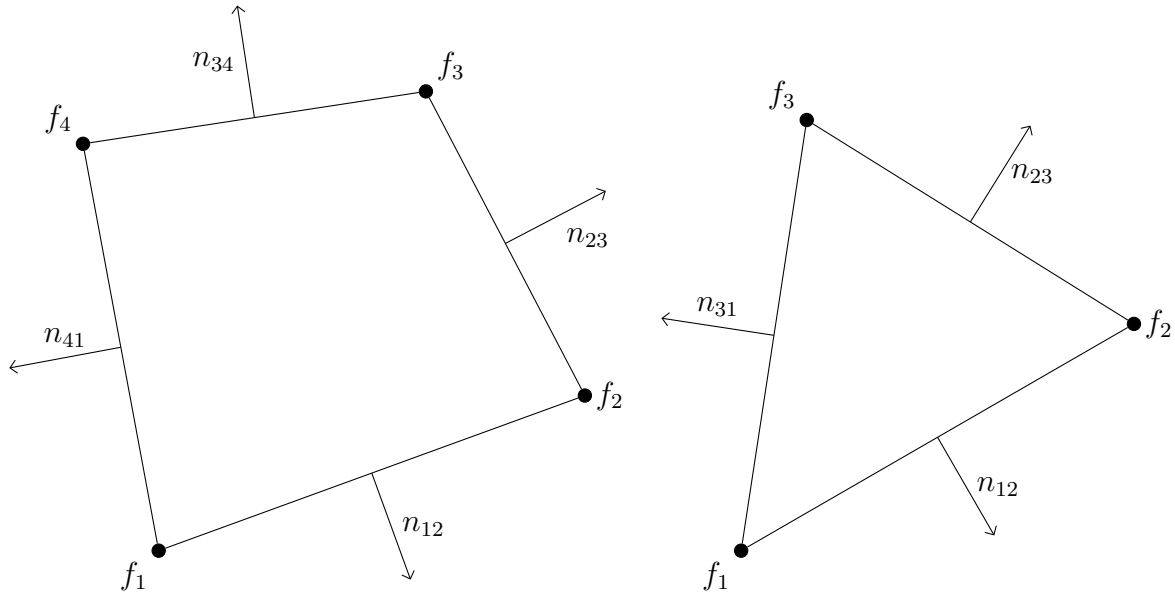

Figure 3.2 Gradient calculation at cells

$$\nabla f_{cell_x} = \frac{n_{12_x} \times \frac{(f_1+f_2)}{2} + n_{23_x} \times \frac{(f_2+f_3)}{2} + n_{34_x} \times \frac{(f_3+f_4)}{2} + n_{41_x} \times \frac{(f_4+f_1)}{2}}{A}$$

$$\nabla f_{cell_y} = \frac{n_{12_y} \times \frac{(f_1+f_2)}{2} + n_{23_y} \times \frac{(f_2+f_3)}{2} + n_{34_y} \times \frac{(f_3+f_4)}{2} + n_{41_y} \times \frac{(f_4+f_1)}{2}}{A} \tag{3.8}$$

Calculation of Gradients at Nodes

The next step in the calculation of the Riemannian tensors is calculating the gradients at the nodes. This is done using the cell gradients and a weighted average method. The cells attached to each node are visited using a node-to-cell connectivity adding the product of each of the components of the gradients and a weighting factor defined by the geometry of each cell. Then, that summation is divided by the summation of the weighting functions of all such cells. Equation (3.9) represents the operations performed for a node depicted in figure 3.3

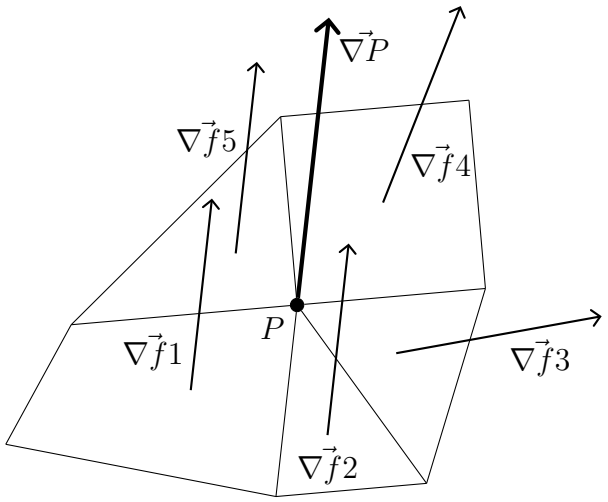

Figure 3.3 Gradient calculation at nodes

$$\nabla f_{node_x} = \frac{f_{xcell_1} \times w_{cell_1} + f_{xcell_2} \times w_{cell_2} + f_{xcell_3} \times w_{cell_3} + f_{xcell_4} \times w_{cell_4}}{w_{cell_1} + w_{cell_2} + w_{cell_3} + w_{cell_4}}$$
$$\nabla f_{node_y} = \frac{f_{y_{cell_1}} \times w_{cell_1} + f_{y_{cell_2}} \times w_{cell_2} + f_{y_{cell_3}} \times w_{cell_3} + f_{y_{cell_4}} \times w_{cell_4}}{w_{cell_1} + w_{cell_2} + w_{cell_3} + w_{cell_4}} \tag{3.9}$$

The weighting function can either be the inverse distance from the node to the centroid of the cells or the area of the cells, and they are selected according to the purpose of the mesh optimization. Figure 3.4 depicts both possibilities.

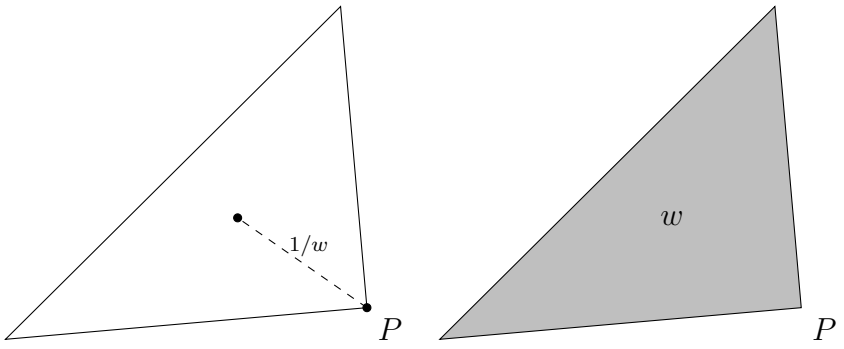

Figure 3.4 Average weighting functions

The algorithm for the gradient calculation can be reviewed using the flowchart provided in figure 3.5.

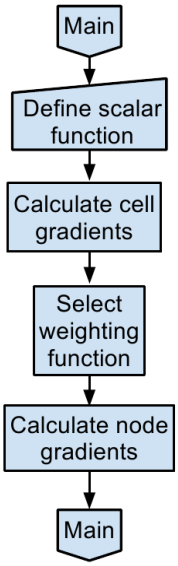

Figure 3.5 Gradient calculation flowchart

Adaptation Functions and Statistics

The next step is to calculate the adaptation functions for each edge connected to each node, and calculate an average and standard deviation. This operation resembles equation (3.6), which for the edges connected to all the nodes looks like (3.10). This information will be useful when making a selection of an adaptation function to be applied over the entire mesh in the desired spacing calculation step.

$$Af_{mean} = \frac{\sum_{i \in nodes} \left[\sum_{j \in edges_i} |\nabla f_{ix} n_{ij_x} + \nabla f_{iy} n_{ij_y}| \times \Delta d_{edge}{}^{P} \right]}{\sum_{i \in nodes} (\#edges_i)} \tag{3.10}$$

Then, the standard deviation is calculated using the same operation defined above for the edges and subtracting from the result the mean average. The root mean square (RMS)

value of Af is computed over all edges. This operation for the edges connected to all the nodes looks like (3.11).

$$Af_{std} = \frac{\sum_{i \in nodes} \left\{ \sum_{j \in edges_i} \left[|\nabla f_{ix} n_{ij_x} + \nabla f_{iy} n_{ij_y}| \times \Delta {d_{edge}}^P - Af_{mean} \right] \right\}}{\sum_{i \in nodes} (\#edges_i)} \tag{3.11}$$

Defining Principal Directions and Desired Spacings

The principal directions are defined using the gradient vector at each node. The first principal direction is obtained by normalizing the gradient vector, and the second principal direction is defined by the unit vector normal to the first.

The spacing in the direction of the gradient is calculated using equation (3.7). First, the user selects a value for the adaptive function to be applied over the entire mesh using the mean adaptive function and the standard deviation. This value is usually set to the mean plus a multiple of the standard deviation [10], as shown in equation (3.12).

$$Af_{threshold} = Af_{mean} + nAf_{std} \tag{3.12}$$

The algorithm for computing the spacing in the second principal direction is defined as twice the average of the length of the edges connected to the node. Figure 3.6 provides a flowchart of the algorithm followed for the calculation of the spacing information.

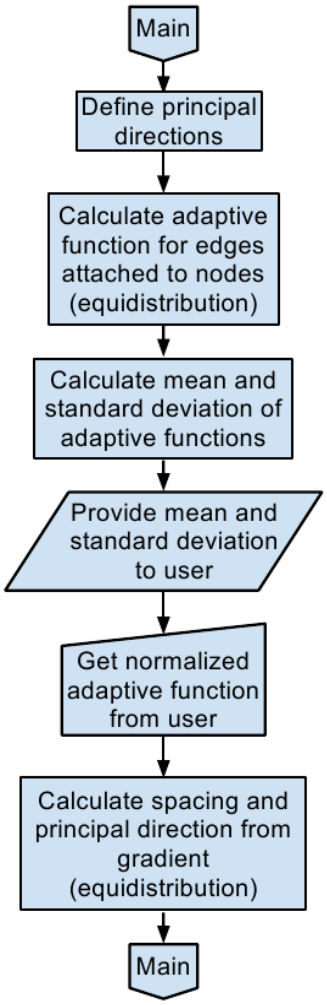

Figure 3.6 Spacing information flowchart

Formulation of The Riemannian Tensors

There is one step left for the calculation of the tensors if the spacing information is to be stored at the nodes. First, the components of the principal direction vectors are placed in the columns of the rotation matrix. Then, the inverse square of the desired spacings are placed in the diagonals of the scaling matrix, and the rest is filled out with zeroes. Then the Riemannian metric tensor is computed, as specified by equation (3.2).

If the spacing information is to be stored at the cells, the tensors for the cells are calculated once the tensors are calculated at the nodes. The tensors belonging to the nodes of the cells are combined using a special function for combining tensors. Such function and others, developed to handle tensors, are presented in the next section.

Functions for the Manipulation of Tensors

A C++ class called `Spacing_Obj` has been developed to handle spacing information. When a spacing field is prescribed for a mesh using a CFD solution, the metric tensor entries calculated for either the cells or the nodes of the mesh are held by `Spacing_Obj`, as well as the tensors' extents. The `Spacing_Obj` class includes functions for manipulating tensors, which are presented in the following subsections. Also, functions for storing and retrieving tensors organized through a quadtree are defined, but this topic is explored in the next chapter. Below are brief descriptions of the functions included in the `Spacing_Obj` class.

Computing Tensors

The tensors are computed using the function `Compute_Riemannian_Metric`. It takes as input the principal direction vectors, and the spacing values in each direction. The eigenvalues of the metric are calculated from the inverse square of the spacing values located in the diagonals of the eigenvalue matrix. The rest of this matrix is filled out with zeroes. The components of the eigenvectors are taken from the components of the principal direction vectors. They are located at the rotation matrix, and a transpose of this matrix is also formulated. The operations performed are a matrix multiplication between the rotation matrix and the eigenvalue matrix, and then a matrix multiplication between the result of the previous operation and the transpose of the direction matrix. The result is a 2×2 matrix, which is returned in an array provided to the function in the argument list.

Calculating Metric Lengths

Calculations of metric lengths are performed by the `Metric_Length` function. It takes as arguments an edge vector and a Riemannian tensor. It performs matrix multiplication between the vector and the matrix, and another matrix multiplication between the result of the previous operation and the transpose of the vector. The square root of the absolute value of the result is calculated and returned, as described by equation (3.13).

$$d_{AB} = \sqrt{\vec{AB}^T [M] \vec{AB}} \tag{3.13}$$

Decomposing Tensors

The decomposition of a metric tensor into its rotation and scaling matrices is performed by the function `Decompose_Tensor`. It takes a Riemannian metric tensor as input and uses the singular value decomposition method to factorize it. It returns the components of the rotation and scaling matrices in two different arrays provided as arguments to the function.

Combining Tensors

The `Combine_Tensors` function is used to take two tensors whose extents may overlap or be adjacent to each other and combine them into a single one. This is the function used to combine the tensors belonging to the nodes of a cell when it is intended to calculate spacing information at the cells. The function takes two Riemannian tensor as input. It decomposes both tensors into their rotation and scaling matrices. Then, it calculates the spacing values from each tensor using the inverse square of the eigenvalues. Next, the tensor with the smallest spacing value is selected in order to compare that tensors' spacings against the inverse of the metric lengths calculated using the selected tensors' principal direction vectors and the other tensor. The `Metric_Length` function is used for this purpose. If

any of the selected spacings is greater than its corresponding metric length inverse, the desired spacings for the new tensor are redefined using the minimum between the selected spacings and their corresponding metric length inverses. With these new spacing values and the principal direction vectors of the tensor selected above, the new combined tensor is calculated using the `Compute_Riemannian_Metric` function.

Comparing Tensors

There are occasions when it is necessary to compare tensors to determine how similar are the metric lengths calculated with one tensor and the principal direction vectors of the other are. It is a useful operation when deciding whether two tensors with extents overlapping or sharing an edge should be combined. The `Compare_Tensors` function can carry out this operation. It takes two tensors as input. The function selects one of the tensors and uses the `Decompose_Tensor` function to factorize it into its rotation and scaling matrices. Then, the values of the spacings are calculated using the inverse square root of the eigenvalues in the scaling matrix. Next, the inverse of the metric lengths are calculated using the principal directions from the selected tensors' rotation matrix and the other tensor. Finally, a ratio is calculated by dividing the differences of the spacings of the selected tensor and their corresponding inverse metric lengths over such spacings. This procedure is repeated, taking the other tensor, decomposing it, and so on. Last, the absolute value of the four ratios are compared and the greatest one is returned. This return value is an indication of the percent difference between the tensors. A value determined by the user (say 5%) or lower is considered small, meaning the tensors are similar.

Converting Scalars Into Tensors

As mentioned before, it is possible to specify spacing at an edge with a scalar. However, when manipulating spacing information involving both tensors and scalars, it is

necessary to transform the spacing information in scalar format to tensor format. The `Scalar_To_Tensor` function can perform this operation. It takes a scalar as input, calculates the scalar's inverse square, and stores it at the diagonal entries of a 2×2 matrix. The off diagonal entries in the matrix are set to zero. This defines a uniform Riemannian tensor, with a constant size in all directions.

CHAPTER 4

SPACING EXTENTS

Definition

Once the tensor or scalar defining the spacing information is calculated, it is necessary to define the extent on which it will apply. In the case of spacing information calculated at the cells, the cells themselves are used as the extents. In the case of spacing information stored at the nodes, the simplest extent that can be defined for an entry is a single cartesian aligned square. Such squares would be defined by a point below and to the left of the node, and by another point above and to the right of the node.

Defining the extents of tensors at nodes using simple extents makes it necessary for extra processing later when retrieving the spacing information. Since the squares are arbitrarily defined by average distances between nodes, there is a chance for overlapping of extents of spacing information at contiguous nodes or existing areas with no information specified. This is undesirable because retrieving spacing information would require testing for overlapping, in which case it would also require testing for similarity between spacing information applying on the overlapping extents.

An option suggested to avoid overlapping of square extents is the use of centroidal duals. They are defined using the centroids of the cells attached to each node. The midpoints of the boundary edges, and the boundary nodes themselves are also used if the extents are calculated for spacing information at boundary nodes. Using the duals as extents of spacing information prevents overlapping of extents and ensures avoiding areas of the grid with no

spacing defined. Figure 4.1 shows samples of both kinds of extents for the same section of a grid.

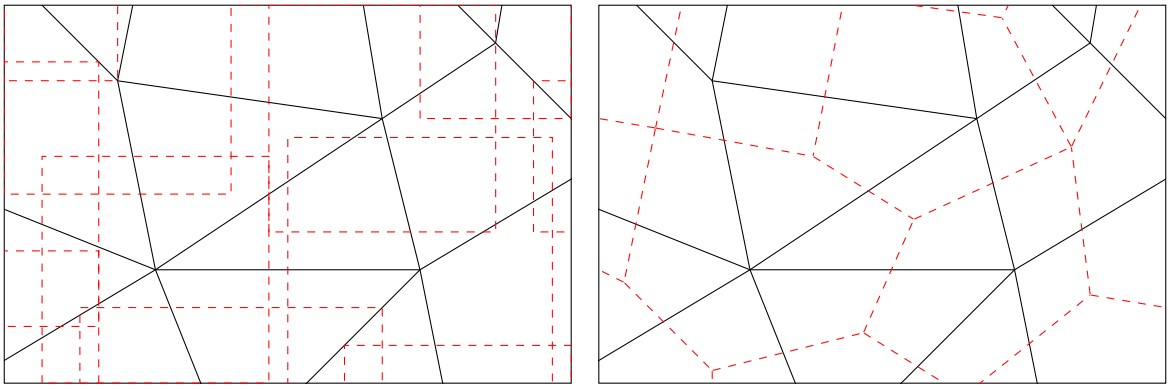

Figure 4.1 Single squares and centroidal duals

Algorithm for Defining the Extents

As mentioned above, one way of defining the extents of spacing information at nodes is by using squares. These squares are defined by two points, whose coordinates are calculated using grid metrics. First, for each mesh node, neighboring nodes are visited using a node-to-node connectivity, and the distance between them is calculated. Then, the average distance to the neighboring nodes is found by dividing the summation of the distances by the number of neighboring nodes. Finally, the coordinates of the points defining the square are calculated by subtracting the average distance from the node's coordinates, and by adding the average distance to the node's coordinates.

The process of defining extents using centroidal duals is more complicated. First, the coordinates of the points used to define the centroids are calculated. The x and y-coodinates of the cells' centroids are obtained by visiting each cell in the mesh and averaging the coordinates of its vertices and storing them in an array. Then, the boundary edges are visited, and the coordinates of the edge's first node and the edge's midpoint are added to the array of the coordinates of the cells' centroids. Also in this step, an edge-to-boundary map is

created to specify the position in the centroids array of the point coordinates obtained from the boundary edges. Previously, the boundary segments containing the boundary edges have to be sorted continually as they appear in the grid so that the last node of one segment is the first node of the next segment in the array of segments.

Next, the dual-to-centroid connectivity is generated. This connectivity serves the purpose of defining the dual cells as a cell-to-node connectivity defines mesh cells. First, memory is allocated for the array to contain the connectivity. The amount of space for the array is calculated by adding the number of cells attached to every node, adding three extra spaces if the node is located in the mesh boundary. Then, each mesh node is visited and treated differently depending on if the node is a mesh boundary node or not. If it is a boundary node, the middle points of the boundary edges attached to the node and that node itself are added first to the connectivity of the dual using the edge-to-boundary map created above. Then, regardless of whether the node is a boundary node or not, the centroidal nodes of the cells attached to the node are added to the connectivity.

The algorithm for generating the extents can be reviewed using the flowchart provided in figure 4.2.

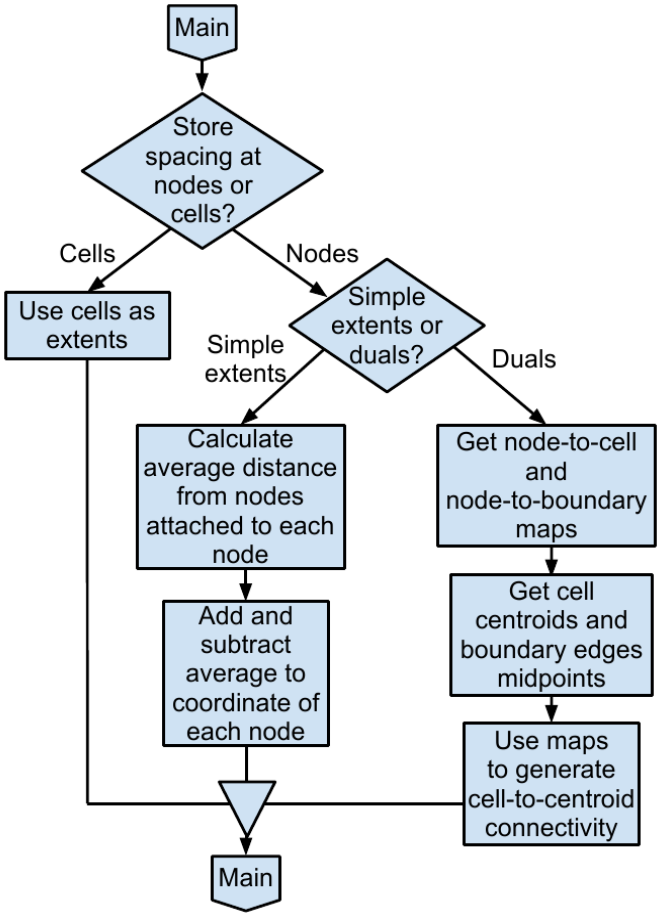

Figure 4.2 Creation of extents flowchart

The Optimize Field Function

The `Optimize_Field` function was written in an effort to improve the efficiency of the process of storing and retrieving spacing information. The idea was to reduce the number of entries in the field by combining the tensors of adjacent extents if the tensors were similar, and by merging such extents into regions. However, a problem was found in the process of developing the function which has not been addressed yet due to time constraints. The algorithm used to define which extents should be merged into regions, and a brief explanation of the problem found are provided below.

Algorithm for Collection of Extents Into Regions

The first part of the optimization process is to define which extents should be merged and collect them into groups. The first step is to define an extent-to-extent connectivity by putting in a list for each extent the indices of the extents that share two vertices with such extent. Then, the extent-to-extent connectivity is used to perform tensor similarity tests between each of the extents in the spacing field and its neighbors. Such tests between two tensors are performed using the `Compare_Tensors` function. The output of the function is compared against a merging threshold specified by the user to define the level of similarity that two tensors should have to be combined. If the `Compare_Tensors` output is smaller than the merging threshold, the tensors of both extents are combined using the `Combine_Tensors` function and the result replaces the previous tensors at such extents.

As tensors are combined across several extents, such extents are given the same tags to later be regrouped. The tags are numbers given to all the extents to be merged into a region. They are given in numbers starting from zero and increasing as new regions are created. The criteria for giving tags is the following:

- If the tensors belonging to two adjacent extents are not similar enough, nothing is done. The similarity is based on the difference between the desired spacing defined by each tensor and the metric lengths calculated using one tensor and the principal directions of the other.

- If the tensors belonging to two adjacent extents are similar enough and neither have a tag, a new tag is created and given to both extents. The tensors are combined, and the old tensors are replaced by the new one.

- If the tensors belonging to two adjacent extents are similar and only one has a tag, the untagged extent is given the same tag of the other. The tensors are combined, and the tensors of all the extents with the same tag are replaced by the new one.

- If the tensors belonging to two adjacent extents are similar and both have different tags, the smaller tag is selected and the selected tag is updated for all the extents having either one of the two tags. The tensors are combined, and similarly, the tensors of all the extents with the selected tag are replaced by the new tensor.

- If the tensors belonging to two adjacent extents already have the same tag, nothing is done.

- Finally, the extents that did not get grouped are given a separate tag.

The algorithm for combining tensors and tagging extents discussed above can be summarized using the flowchart provided by figure 4.3.

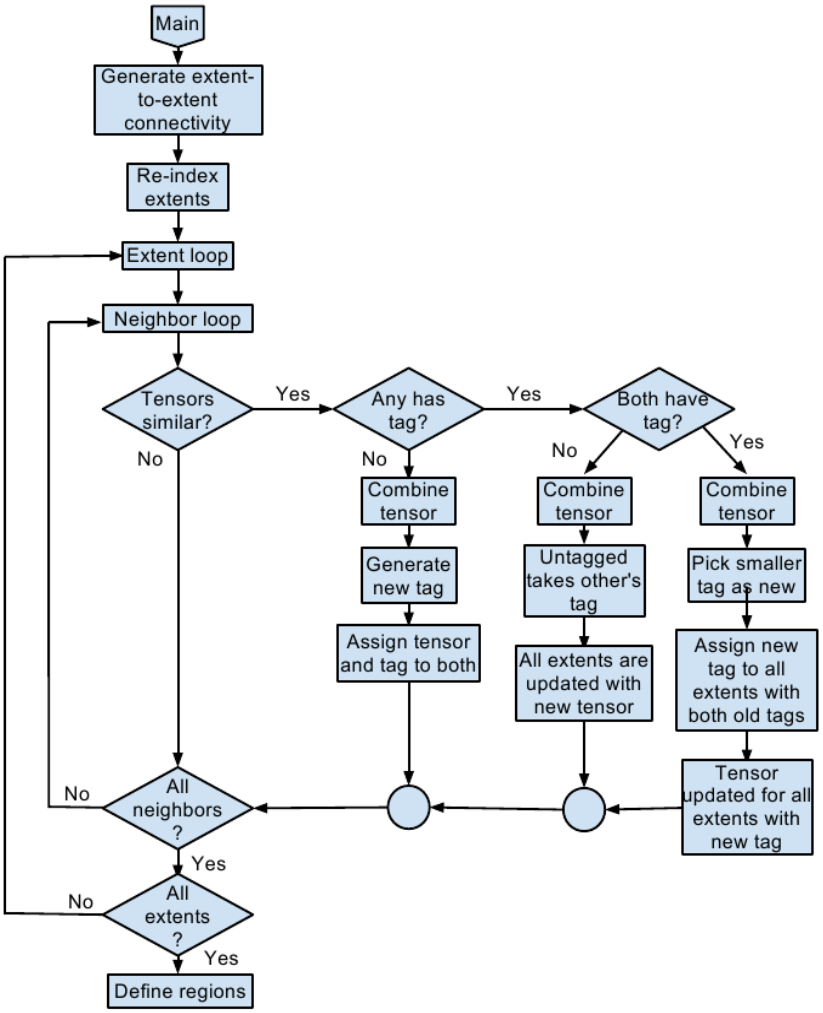

Figure 4.3 Combine tensor algorithm

The second part of the optimization process is to define the connectivity of the regions to the vertices that define them. This connectivity would work similar to the dual-to-centroid or the cell-to-node connectivities except for the fact that the formed regions are not duals or cells anymore. First, the indices of the extents forming the regions are listed into an array per region. Then, an account is kept of how many times the vertices defining the extents are used by the extents in a region. The account is generated by visiting all the extents in a region, by visiting the vertices of the extent, and by increasing the account of each vertex by one. Then, this account for each vertex is compared to the number of vertices connected

to such vertex. If both numbers match, the vertex is marked for deletion. If the numbers do not match, they will become boundary nodes of the region.

The next part of the optimization process is to sort the vertices of the region boundary consecutively. First, a list of edges is generated by visiting each extent in a region and adding each of the edges that define it to the list. Then, the edges go through a selection process to see which ones will stay to form the region boundary. If an edge is connected to a node marked for deletion, then the edge is also marked for deletion. If an edge is connected to a vertex which is used twice by extents in the region, it is searched for again to see if it repeats in the list. If it does, it is marked for deletion. Then, the edges are sorted by finding the second node of one edge as the first node of another, successively until returning to the initial node. Finally, consecutive edges that align are joint into a single edge. The alignment test is done, first, by calculating the edges unit vector. Then, the absolute value of the components of the vectors of consecutive edges are compared. If they match, the edges are joint together.

The algorithm discussed above, used for defining region connectivity, is summarized by the flowchart presented in figure 4.4.

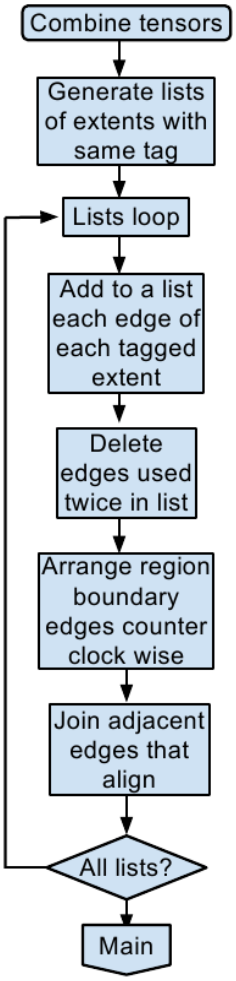

Figure 4.4 Algorithm for defining connectivity of regions

A problem has been encountered with this last part of the function implementation. Multiply bounded regions have been generated which do not allow defining connectivity for some regions in some of the test cases used. A discussion on this is included in Chapter 6.

CHAPTER 5

QUADTREE STORAGE

Definition

A tree data structure is a data organization model that resembles a hierarchical tree structure in which information is represented using parent-child relationships [11]. Trees can be used to organize many kinds of data. In this work, quadtrees are used to organize metric spacing objects by storing pointers to objects containing spacing and extent information. The purpose is to speed up the process of retrieving spacing information based on the location of the information in the mesh and not in an array.

In the application of spacing fields, a quadtree is defined by the rectangular extents, called root quadrant, containing the spacing field components to be organized and by the four equally subdivided regions, called quadrants, of the extents. Such regions may in turn be succesively subdivided into four other quadrants depending on the criterion used to store information. If an extent is subdivided, it is called "the mother" of the subdivisions, which are called "the children". The set of new extents resulting from the same number of subdivisions are called a layer. Therefore, the root quadrant is said to be in the first layer, the four extents or quadrants of the first subdivision are said to be in the second layer, the sixteen extents resulting from the second subdivision are said to be in the third layer, and so on.

Algorithm for Storing Spacing Information

The process of loading information on the tree starts with the definition of a box for each spacing entry, which contains the extent of the spacing information as shown in figure 5.1. This is done by visiting the vertices that define the spacing extent and selecting the left-most and right-most x-coordinates, and the highest and lowest y-coordinates.

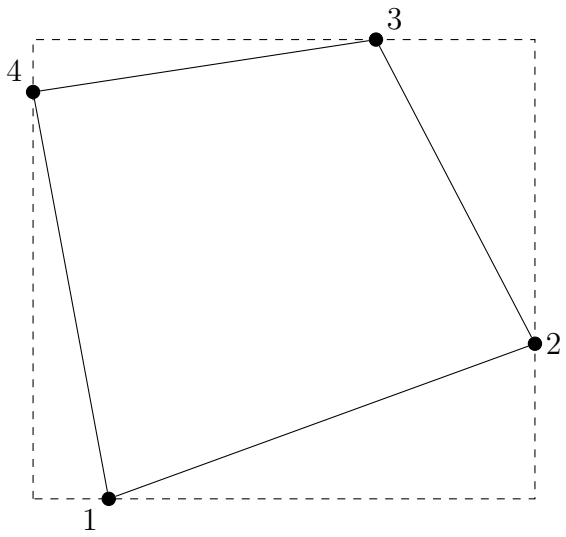

Figure 5.1 Extent box for entry in quadtree

The next is a recursive step in which pointers of `Spacing_Obj` objects are stored in quadrants at different layers depending on their box extents. First, the coordinates of the middle point of the quadrant are calculated to trace imaginary horizontal and vertical lines passing through the middle point and dividing the quadrant in four. Then, the extension of each box is tested against the coordinates of the middle point to see whether the spacing extent lays on the imaginary lines, and if not, to find out in which of the subdivisions it falls. This is depicted by figure 5.2.

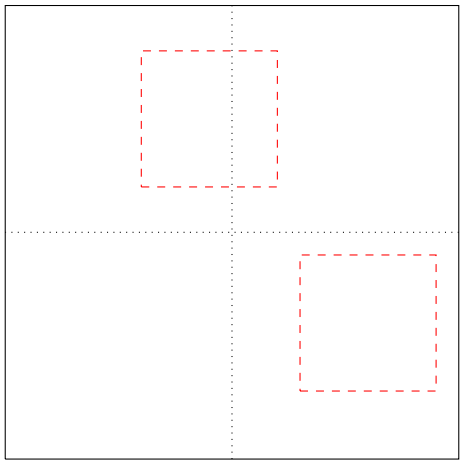

Figure 5.2 Extent box location test

The pointers of the spacing objects whose extents fall on the imaginary lines get stored on the present quadrant. The pointers of the rest of the objects get collected in one of the four sets arranged according to the quadrant where their spacing extents fall. These lists are passed to the function executing this process on the current quadrant, to execute on each of the children. This step is repeated again for each of the quadrants. The quadrants are subdivided and information is passed further up the tree until all the remaining extents lay on the dividing imaginary lines of some subdivision of the quadtree at some higher level, unless a maximum number of layers is specified. In that case all the remaining pointers are stored at the specified maximum layer.

The algorithm for storing information can be followed using the flowchart provided in figure 5.3.

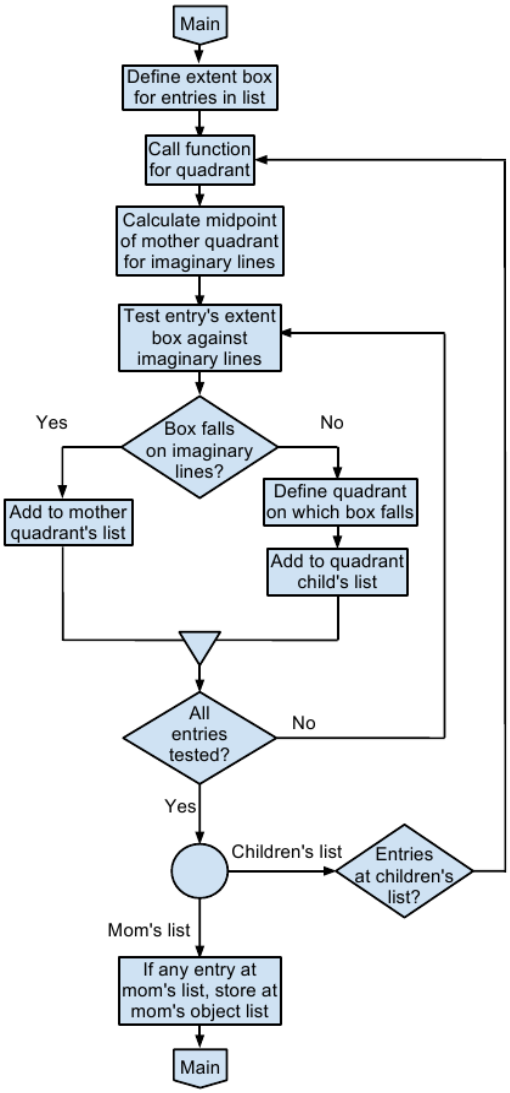

Figure 5.3 Algorithm for storing information in tree

Algorithms for Retrieving Spacing Information

Spacing Information at Points

The first step in retrieving spacing information from the tree is to get a list of spacing objects whose extents may contain the position of the point. This is done by adding the objects stored at the first layer to a list, and by identifying the subdivision quadrant where the point is located. Then, the objects stored at the identified quadrant at the next layer

are added to the list, and the subdivision quadrant where the point is located is identified again. This process is successively repeated until arriving to the last subdivision's quadrant where the point is located.

The next step is to identify which spacing's extents actually contain the location of the point. This is done by using the location of the point and the location of the vertices of each extent in the list. The normal vector to each edge of the extent pointing out of the extent is dotted with the vector pointing from the middle of the edge to the location of the point, figure 5.4. If the dot product is negative for all the extent edges, the extent contains the location of the point.

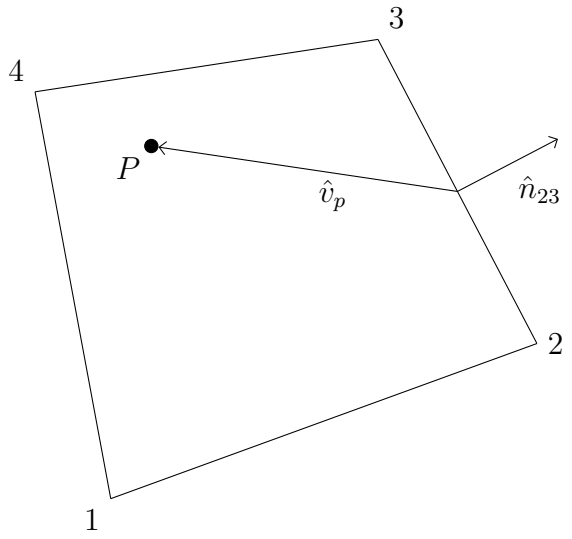

Figure 5.4 Point in element test

Finally, the actual spacing information is obtained for the location of the point. If the spacing information requested must be defined by a scalar, the minimum of the scalar spacings, specified at the extents containing the point, is selected. If any of the extent's spacing information was defined using a tensor, the tensor must be decomposed, the desired spacings must be obtained from the entries in the scaling matrix, and the minimum spacing

should be considered for candidate to be returned as spacing information at the point. If the spacing information requested must be defined by a tensor, the tensors of the extents containing the point are combined. If any of the extent's spacing information was defined using a scalar, the scalar must be transformed to a tensor, and it should be combined with the other extents' tensors.

Spacing Information for Edges

Again, the first step in retrieving spacing information for an edge is to get a list of spacing objects whose extents may contain a piece of the edge. In order to do this an extension box of the edge is defined with the middle point of the edge as the middle point of the box and the length of the edge as the box sides dimensions. Then, the objects stored at the first layer are added to a list, and the subdivision quadrants are tested to see if they contain any piece of the edge extension box. If they do, the objects of those quadrants at the next layers are added to the list, and the test is done again on the quadrants at the next layer. This process repeats successively until all the spacing objects stored at the subdivisions containing any piece of the edge's box are retrieved.

Next, the extents containing a piece of the edge are identified, and the portion of the edge they contain is calculated. The edges of the extents are visited, and dot products are calculated between the extent's edges normal vectors pointing out of the extent and the vectors pointing from the middle of the extent's edge to the ends of the edge for which information is being retrieved, figure 5.5. If both dot products are positive for all the extent's edges, the extent contains no piece of the edge for which information is being retrieved. If for any edge one of the dot products is negative and the other is positive, the edge for which information is being retrieved is cut at its intersection with the extent's edge. Once, the edge has been resized to the extent's edges it bisects, the length of the contained piece of the edge is calculated.

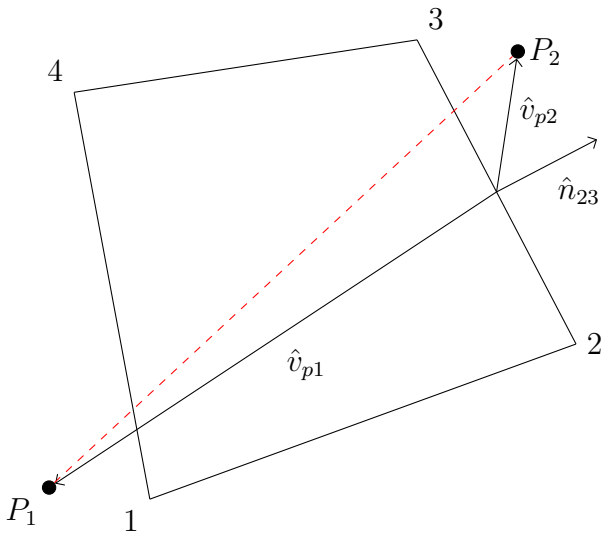

Figure 5.5 Edge in element test

Finally, the spacing information is obtained for the entire edge. If the spacing information requested must be defined by a scalar, a weighted average of the scalar spacings, defined at each of the extents the edge intersects, is calculated. The weighting function used for averaging is the portion of the edge contained by the extents. If the spacing information at any of the extents was defined by a tensor, such tensor must be decomposed, the desired spacings must be calculated from the entries of the scaling matrix, and the minimum desired spacing should be used for the averaging of the scalar spacing of the edge. If the spacing information requested must be defined by a tensor, the tensors of the extents containing pieces of the edge must be combined.

The algorithms discussed in this section are presented in flow charts in figures 5.6, 5.7, and 5.8.

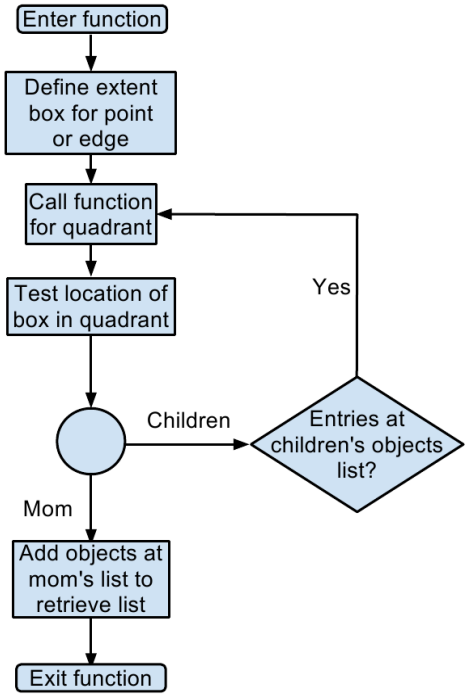

Figure 5.6 Algorithm for retrieving list of items from quadtree

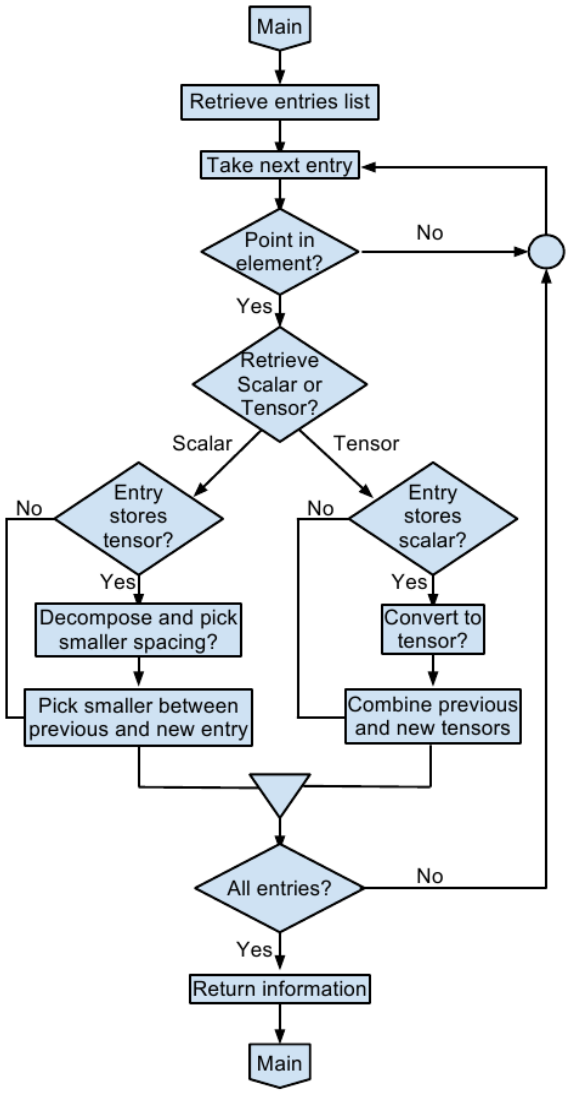

Figure 5.7 Retrieving spacing information at a point

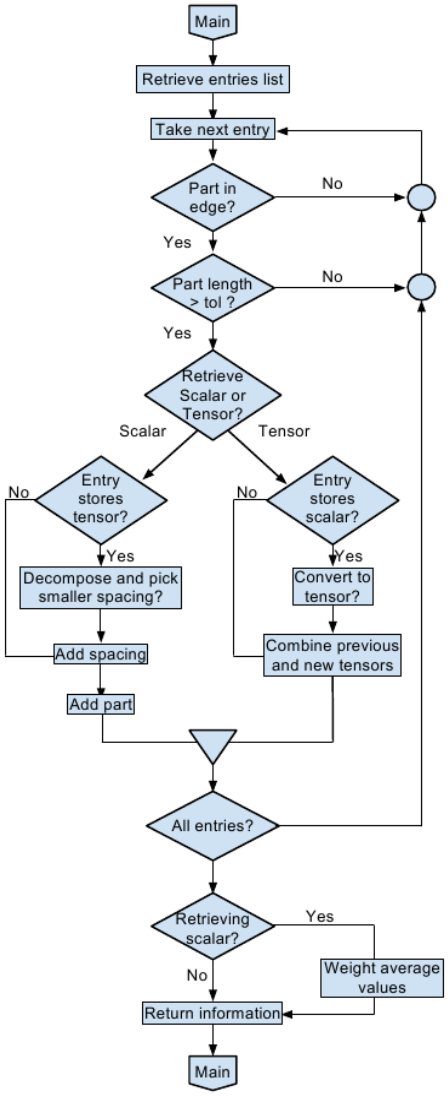

Figure 5.8 Retrieving spacing information for an edge

The Quadtree Class

A C++ class called `Quadtree_Storage` has been developed to handle pointers to the spacing objects and properly locate them in a quadtree. The class holds information on the extents of the quadrant the class' object prescribes, and on the level to which the quadrant belongs. It defines pointers to the class object itself and to the four `Quadtree_Storage` objects of the children that would be generated if the objects' quadrant was subdivided.

The class also defines functions for storing and retrieving data. They are presented in the following subsections.

Function for Storing Objects

An overloaded `Quadtree_Storage` function called `Store_In_Quadtree` was defined to either store one pointer in the tree, or to store an array of pointers.

In the first case, the function finds the coordinates of the middle point of the quadrant. Then, using the extent box of the pointer to be stored, the function finds out whether the pointer's box falls on any of the imaginary horizontal or vertical lines that pass through the quadrants middle point, or not. If it does, the incoming pointer is added to the list of items for that quadrant. If not, the function finds out on which subdivision quadrant the pointer's box falls, and recursively calls the `Store_In_Quadtree` for the quadrants child.

In the second case, when pointers in an array are to be distributed in the quadtree, the function finds the coordinates of the quadrant's middle point through which imaginary horizontal and vertical lines pass. Then, the function tests the boxes of the incoming pointers to see which fall on the imaginary lines, and which fall on any of the subdivision quadrants, keeping a count of the number of pointers per quadrant. This information is used to prepare lists for the pointers that fall on the different quadrants. Then, the function tests the boxes again, but this time to add to the items list the pointers whose boxes fall on the imaginary lines, and to add the other pointers to the list of the appropriate quadrant they correspond. This process of selecting the items that get stored in the quadrant of the current layer is depicted by figure 5.9. Finally, the function `Store_In_Quadtree` is recursively called for the children passing to each their corresponding list of pointers and pointer boxes.

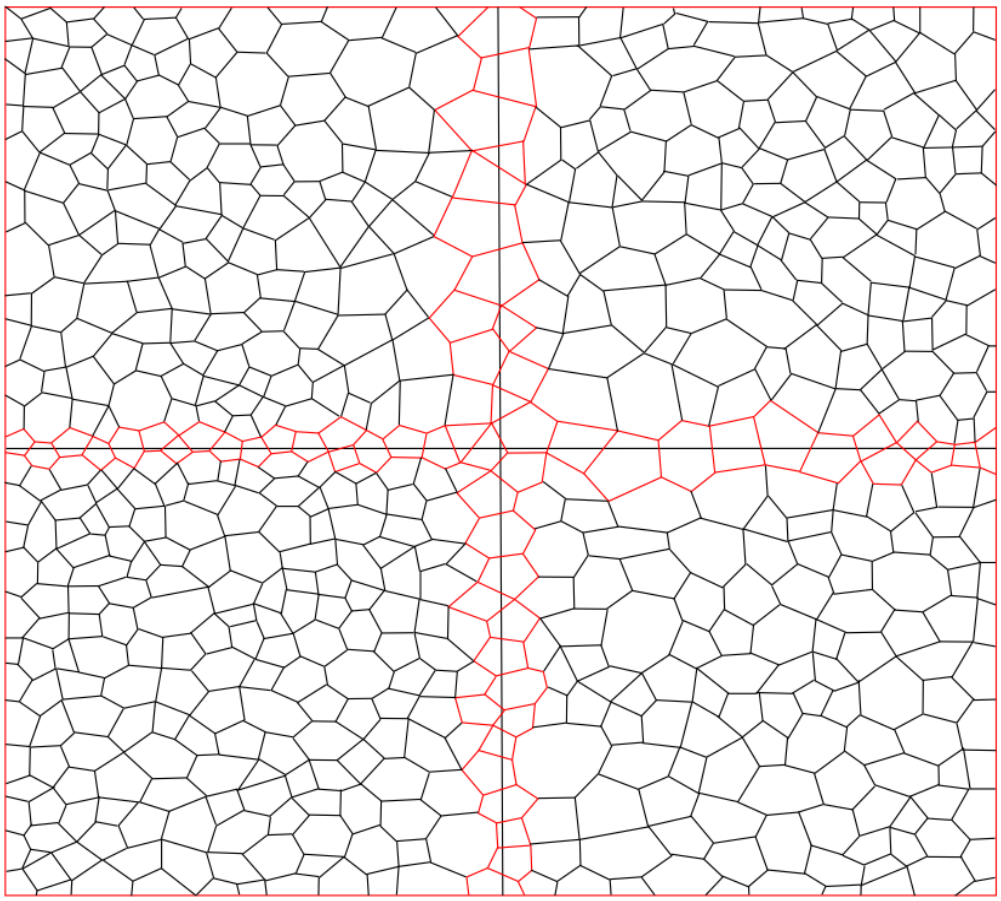

Figure 5.9 Elements staying at layer

Function for Removing Objects

An overloaded `Quadtree_Storage` function called `Remove_From_Quadtree` was defined to either remove a pointer by specifying the pointer or to remove a pointer by specifying the pointer and a box containing it.

In the first case, the pointer is searched for in the array of items of the object in order to be deleted. If it is found, the item is deleted. If the items array of the subdivision quadrant is empty after deleting the pointer given to the function, the object itself for that quadrant is deleted making sure it does not belong to the root quadrant (the object whose

extension is not the one covering the entire spacing field). If the pointer is not found, the `Remove_From_Quadtree` is recursively called on each of the quadrant's children.

In the case when the pointer's box is specified with the pointer, the function finds the coordinates of the quadrants middle point through which imaginary vertical and horizontal lines pass. Then, the incoming pointers box is tested against the middle point coordinates to see if it falls on the imaginary lines or not. If it does, the pointer is searched for in the object's items array to be deleted. Then, the object is deleted if the array is left empty making sure the object does not belong to the root quadrant. If the incoming pointer's box does not fall on any of the imaginary lines, the `Remove_From_Quadtree` function is recursively called for the child of the quadrant where the box falls.

Function for Retrieving Objects

A function called `Retrieve_List` is defined for retrieving a list of items possibly related to a point. The function takes as arguments the coordinates of the point and tolerance values for the x and y-dimensions. The tolerances are very small dimensions relative to the dimensions of the spacing field (usually one over thousandth of the lengths of the spacing field in the x and y dimensions).

The function proceeds to define a box for the given point by adding and subtracting from the points coordinates their corresponding values of the tolerances. Next, the location of the box is tested to see if it is inside the quadrant extension. If not, no items are returned. Otherwise, the `Retrieve_List` function is recursively called for the children collecting items in a list. Once the functions return from the children, the items held by the quadrant are also added to the list of items to be returned.

Function for Replacing Objects

The function `Replace_In_Quadtree` was written to replace a pointer with another one by specifying the pointers box. Again, the function finds the quadrant's middle point coordinates through which imaginary vertical and horizontal vertical lines pass. The location of the box is tested to see if it falls on any of the imaginary lines, and, if not, to locate the quadrant on which it falls. If it falls on any of the imaginary lines, the pointer is searched for in the quadrant's items list, and if found, it is replaced as well as its extent's box. If the incoming pointers box falls on a quadrant, the function `Replace_In_Quadtree` is recursively called for the child at that quadrant.

Functions for Retrieving Other Information

A function for retrieving the extents of the root quadrant called extents was created. Another function called `max_level`, written to find out the maximum level in the tree, visits every single subdivision of the root quadrant and returns the maximum level on which a subdivision may be located. Similarly, a `finest_size` is a function created to visit every single subdivision to find out the smallest extension that a subdivision may have.

Functions for Handling Spacing Information

Function for Storing Spacing Information

A function called `Populate_Quadtree` is defined for the `Spacing_Obj` class in order to organize the scalars or Riemannian tensors in a quadtree. This function takes the addresses of the objects containing information on the components of the spacing information and the vertices of the extents on which they apply, and copies them to an array of pointers. It also defines the extents boxes of the extents and stores this information in arrays. Then, it calls the `Store_In_Quadtree` function of the `Quadtree_Storage` class, and provides the array of

pointers, and the array with information of the extents boxes of the spacing information extents.

Functions for Retrieving Spacing Information

The `Spacing_Obj` class also defines functions for retrieving spacing information. The `Retrieve_Scalar` and `Retrieve_Tensor` functions return spacing information for the location of a point. The arguments for these functions are the coordinates of the point, and, as indicated by their names, the first function returns a scalar and the second one returns a tensor. Similarly, the `Retrieve_Edge_Scalar` and `Retrieve_Edge_Tensor` functions are defined for retrieving spacing information for edges. The arguments for these functions are the coordinates of the points that define the edge.

Other Functions

As seen in the algorithms subsections for retrieving information, it is often necessary to find out whether the location of a point is contained inside an extent. This task is performed by the `Point_In_Element` function, which takes as argument the coordinates of the point and a pointer to the object holding information on the vertices of the extent. It returns one if the location of the point is found inside the extent, and zero otherwise.

The calculation of the length of an edge contained inside a extent, if at all, is carried by the `Edge_In_Element` function. It takes the coordinates of the points defining the edge and a pointer to the object holding information on the vertices of the extent. If the value returned is zero, no portion of the edge is contained by the extent.

The `Compute_Edge_Metric_Length` function was defined to compute metric lengths for edges. The spacing information retrieved from each extent containing a portion of the edge is used to compute the metric length for such portion. If the spacing information at an extent is defined by a scalar, the metric length is calculated by the length of the entire

edge over the spacing at that extent. If the spacing information at an extent is defined by a tensor, the edge vector is defined and the metric length is calculated from such vector and the tensor using the `Metric_Length` function. The `Compute_Edge_Metric_Length` gives the option of returning the maximum of the metric lengths calculated at each extent, or returning a weighted average of such metric lengths using the length of the portion of the edge at each extent as the weighting function.

The `Retrieve_Tensor_Item` function was defined to retrieve information from a specific object. The function takes the index of the object in the list of objects and returns the spacing information in the form of a tensor, the number of vertices of the extent, and the coordinates of such vertices.

Factors that Impact the Efficiency of Retrieving Spacing Information

Combining Information

As described in the above section on algorithms for retrieving information at a point, the algorithm takes care of combining spacing information from extents containing the location of a point. Having more than one extent containing the same point happens due to square extents overlapping. This issue slows down the process of arriving to the requested spacing information since it makes necessary executing extra operations like selecting between spacing values, combining tensors, and converting scalars into tensors and vice versa. This source of inefficiency can be avoided by using centroidal duals as the extents of spacing information at mesh nodes. Duals ensure avoiding overlapping extents, which in turn ensures finding the location of a point at a single extent.

Number of Entries in the Spacing Field

It was observed that the algorithm retrieves a list of entries to test their extents for the location of a point or an edge. The number of entries in this list is relatively large with

the number of entries that may actually contain the location of the point. This is due to the fact that the algorithm adds all the entries on a quadrant at different layers until reaching the last subdivision of the root quadrant where the point may be located. Because of this, the retrieved lists are larger for larger meshes since the extents are defined using mesh cells or centroidal duals around nodes.

The process of testing extents for location of nodes and calculating portions of edges in extents is computationally demanding. Therefore, the number of entries in a spacing field is a very important factor in the efficiency of retrieving information. Large numbers of entries slow down the process of finding desired spacings, so the smallest necessary number of entries is desirable.

Solving this issue was the motivation for developing the `Optimize_Field` function. As described in Chapter 4, the task of the `Optimize_Field` function was to merge extents into regions based on the similarity of the spacing information defined on them. Unfortunately, there are still issues in the algorithm that will be addressed in future research work. More on this is provided in the chapter on further research.

Storing Information Over a Large Area on the Same Layers

Currently, the algorithm for storing information stores on the quadrant of the first layer objects whose extents fall on the imaginary lines that divide the quadrant in four. Then, the algorithm visits the subdivisions of the next layer and does the same on each subdivision, and so on. The algorithm for retrieving information looks, at each layer, for the quadrant containing the requested location of information, and goes to that quadrant in the next layer. As it visits the quadrants at different layers, it adds to a list all the objects stored at those quadrants. From this brief summary of both algorithms, it is seen that the pattern of storing information of the first one counteracts the pattern of retrieving information of the second. While one stores information spread over a quadrant, the other seeks to gather

information only from the subdivision of the quadrant where the requested information is located. This causes gathering of a lot of information very unlikely to be used, which is a problem similar to the one presented in the previous subsection. No work has been done for fixing this issue yet, but ideas for solving it are presented in the further research chapter.

CHAPTER 6

THE CREATE-SPACING PROGRAM

Introduction

A program called `create_spacing` has been created with the purpose of generating a spacing field from CFD solution data obtained on a 2D hybrid mesh (triangles-quadrilateral). The grid information and the solution data are provided through a generic mesh file. The spacing information can be calculated for nodes or for cells and can be defined through tensors or scalars. Also, the program defines the extents on which the information applies. The spacing information and the definition of the extents are exported to a tensor file. Along with the tensor file, the program generates a VTK file [12]. The VTK file allows the user to visualize the physical extents of the spacing information and the magnitude of the spacings at each extent. The program provides the user with the option of storing solution data, such as the velocity magnitude and Mach number, in the tensor file instead of spacing information. If the user chooses this option, no VTK file is generated.

Input File

The mesh information and CFD solutions should be provided to the program through a generic mesh file (.mesh extension). The mesh information provided should be the coordinates of the nodes, element connectivities of the triangle and quadrilateral cells, and the boundary connectivities. The CFD solutions provided should be node based, and should be four: density, x-momentum, y-momentum, and total energy.

Output Files

Tensor File

The program exports the information generated to a file called the tensor file (.tensor extension). The tensor file specifies the number of entries, the number of vertices per entry extent, the coordinates of such vertices of the extent, and the components of the metric tensors. If the spacing information at an extent is specified by a scalar, such scalar is specified instead of a tensor.

Early in the execution of the program, the user is given the option of exporting solution data to the tensor file instead of spacing information. If this option is chosen, the user is given four options to choose from: velocity magnitude, pressure, mach number, and density. After the user picks an option, the program creates the centroidal duals as extents for the solution data since the solution data is node based. Then, it exports these to the tensor file.

Visualization File

The program generates a VTK file (.vtu extension) for visualization purposes [12]. It contains information on the extents which are the cells themselves if the spacing information is stored at the cell, and the centroidal duals if the information is stored at the nodes. The file also contains data for visualization of tensors and spacing vectors. The following pictures are obtained from the VTK files using the Visit software for visualization [13].

Running The Program

The program is run by typing the name of the executable (`create_spacing`) in the terminal, and the only argument it takes to run is the name of the mesh file. As the program executes, it shows in the screen the operations it is performing and asks the user to provide input based on the options presented.

The first option the user is asked to select is the solution data the program should use as the analytic function, on which the calculation of spacing information will be based. The options are the velocity magnitude, pressure, mach number, and density. The user should type in the integer representing his or her choice.

Next, the user is given the option to store in a tensor file either solution data previously calculated or spacing information. If the user chooses to store solution data, it is only stored node based and the extents exported are the ones of the centroidal duals. The user is asked to type in a name for the tensor file (should have .tensor extent), no VTK file is generated, and the program exits successfully.

If the user chooses to store spacing information, the program proceeds to calculate the cell gradients. Then, it proceeds to average those gradients for the nodes using a weighting function. The user is asked to choose whether to use the area of the cells surrounding the nodes, or to use the inverse distance to the centroid of such cells as the weighting function.

The next step is the calculation of statistics of the analytical function. The adaptive function mean and standard deviation are provided to the user. This information is used to select a threshold value of a limit for the adaptive function used later in the calculation of Riemannian metrics. The user is also asked to input a value for the power of the length. More information on these parameters have been provided in Chapter 3.

Then, the program prepares to calculate spacing information, defined through tensors or scalars, and export it to a tensor file. The user is asked to select whether to store the information node based or cell based, and whether to store the information as scalars, as a tensor, or as a combination of both depending on how big the magnitude of spacing in one direction is with respect to the other (if one is bigger than the other by less than 50%, the information is stored as a scalar). Depending on the users choice, the program proceeds to calculate spacing information and extents on which they apply. Before storing all these, the spacings generated greater than a threshold calculated based on grid metrics (one over tenth

of the largest dimension of the entire grid) are filtered out. Then, it asks the user to enter names for the VTK file and tensor file. Then the program exits successfully.

The algorithm followed by the program can be summarized through the flowchart provided in figure 6.1.

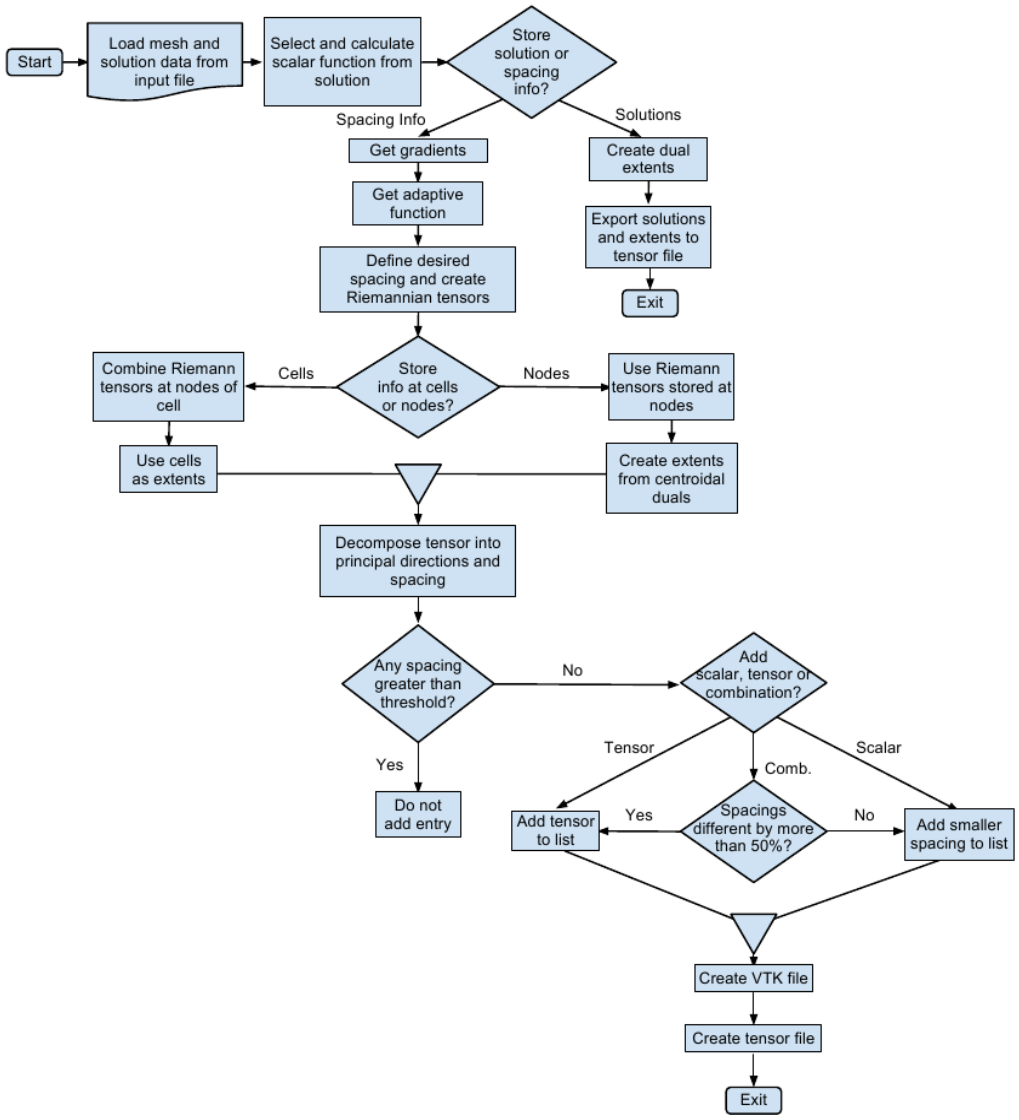

Figure 6.1 Algorithm of the `create_spacing` program

Test Cases

The first test case presented here is a square mesh of squares. It was a simple case used to test the calculation of the gradients and the spacing along the gradients. No CFD solutions were used. The mesh was equally divided in three regions shaped as vertical rectangles. The scalar function was set to zero for the nodes of the left region, and to one for the region at the right. The scalar function for the nodes of the middle region were set between zero and one depending on their location in the x-direction.

Next, two meshes of a ramp at a 30 degree angle were used for testing and developing the program. The reason for using the 30 degree angle was to make sure a shock would be obtained, and therefore, a region with large gradients to be used in the calculation of spacing information. The first mesh is an all triangle mesh on which CFD solutions were obtained for air flow moving at Mach 2. The solver program used for obtaining the solutions was developed by Taylor Erwin [14]. The second mesh is hybrid (triangles and quadrilaterals) and was only used for testing the code when generating centroidal duals. No CFD solutions were obtained for this mesh due to the unavailability of a working 2D solver for hybrid meshes.

Visualization of Results

The following are visualizations of the results generated by the `create_spacing` program and exported to VTK files. The images were obtained using the VisIt software [13].

Square Mesh - Validation of the Spacing Field

The create program was run on the square mesh with the main purpose of checking the calculation of gradients and spacing is performed correctly. Figure 6.2 shows the plot of the scalar function for this mesh, defined in the test cases section, using the VisIt software.

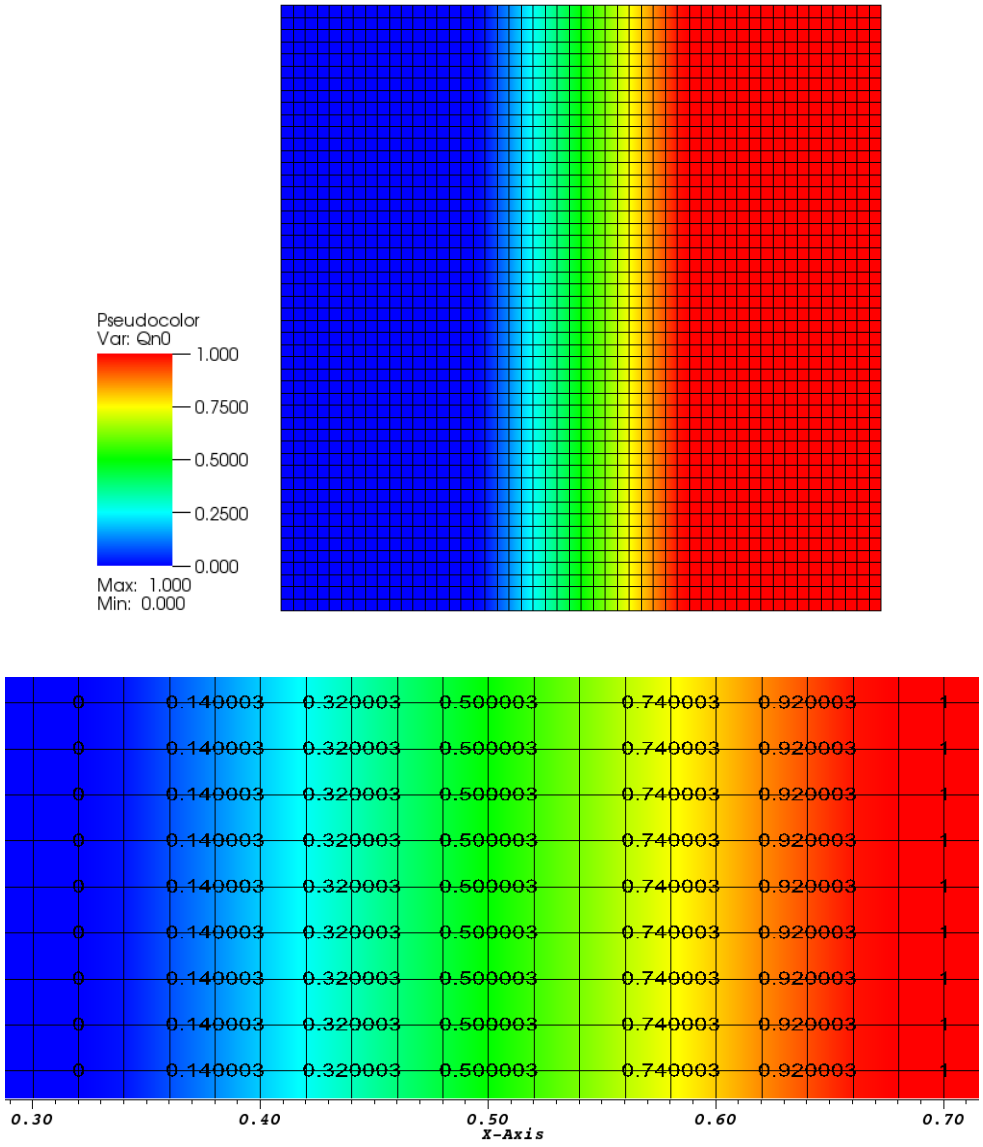

Figure 6.2 Scalar function for square mesh

The gradient across the middle region was manually calculated using a finite difference method, as shown in equation (6.1). Since the scalar function changes only in the x-direction, the gradient vector is expected to point only in the positive x-direction, as well, and with magnitude 3. The plot of the gradients in a dual mesh (spacing information stored at nodes) in figure 6.3 agree with the result of the operation discussed above.

$$\frac{df}{dx} = \frac{f_{right} - f_{left}}{x_{right} - x_{left}} = \frac{1 - 0}{0.666 - 0.333} = 3 \tag{6.1}$$

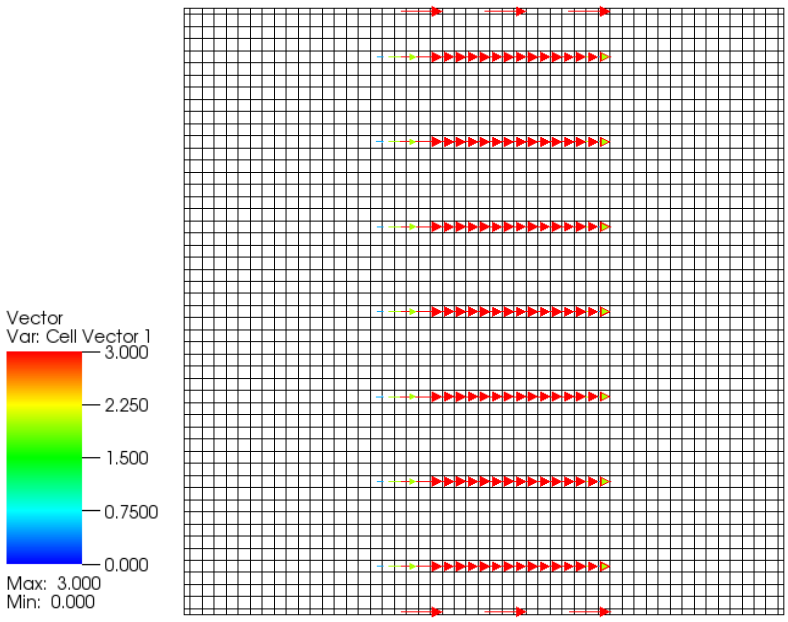

Figure 6.3 Plots of gradients for simple test case

The plotted vectors showing a magnitude less than 3 are result of the averaging of the gradients at cells in the calculation of gradients at nodes, as discussed in Chapter 3.

As the program ran, the parameter P, described in Chapter 3, was set to 1. The adaptive function threshold was set to the mean value of the adaptive function across the mesh, which was 0.01. With these parameters and using equation (3.7) the value of the spacing in the middle region is expected to be 0.00333. The spacing along the direction of the gradient calculated by the program checks with the expected value, as shown by the plot of the spacing in figure 6.4.

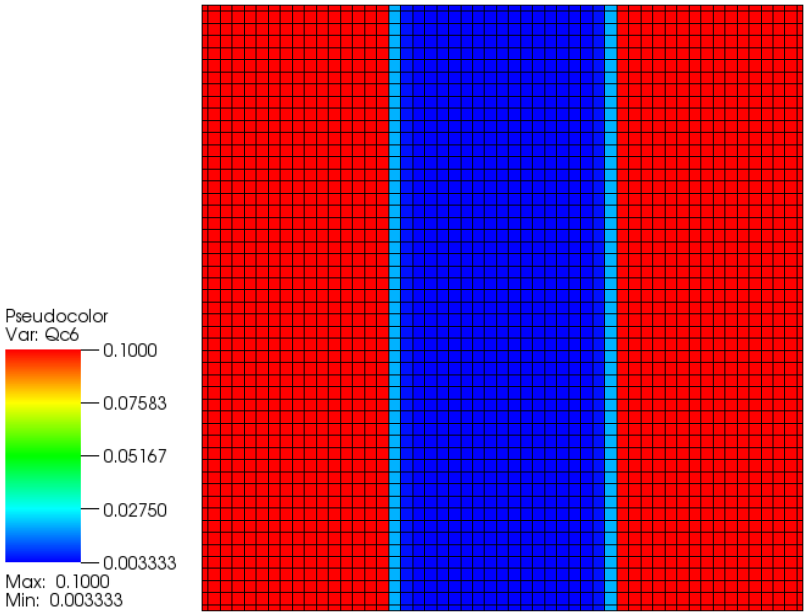

Figure 6.4 Plot of the spacing along the direction of the gradient vector

Again, the plotted spacing values showing a magnitude greater than 0.00333 and less than 0.1 are result of the averaging of the gradients at cells in the calculation of gradients at nodes, as discussed in Chapter 3. The values of the spacing at the left and right regions were set to one tenth of the maximum dimension of the entire mesh, as discussed in the section of the algorithm of the program. The purpose is to avoid getting large or infinite spacing values in regions where the gradient is zero or nearly zero.

Hybrid Mesh

The main purpose of using this test case was to test the algorithm for the generation of dual extents. As mentioned in the test case section, no CFD solutions were obtained using this mesh. The mesh was generated using the Pointwise software, and is shown in figure 6.5. The quadrilateral cells were located in a position where a shock would be expected if an air flow approached the ramp at Mach 2.

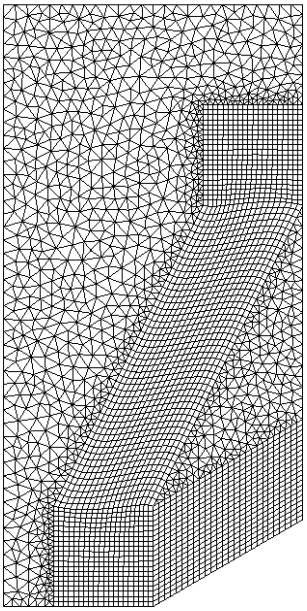

Figure 6.5 Hybrid mesh

Figures 6.6 and 6.7 are images of the mesh obtained using the centroidal dual extents and an overlay plot of a section of the original mesh and the centroidal duals. A plot of the extents of the spacing information saved at the cells is not provided since such extent mesh is identical to the original mesh.

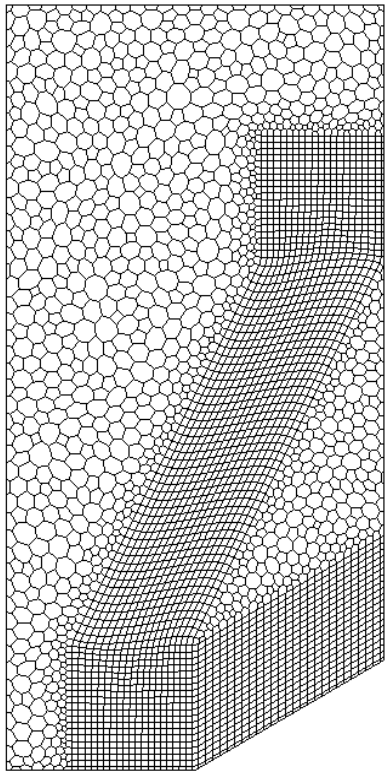

Figure 6.6 Dual mesh for hybrid mesh

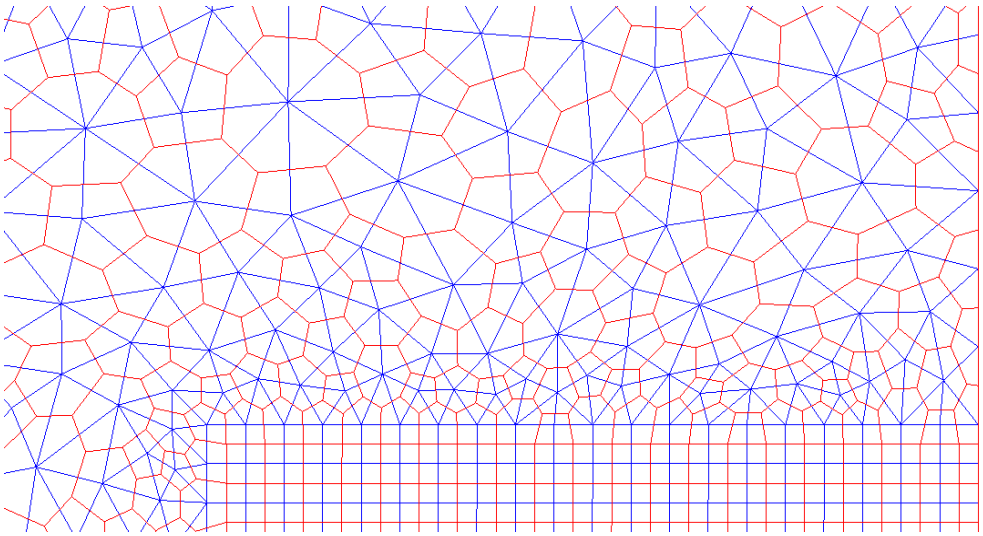

Figure 6.7 Overlap of dual and hybrid meshes

Triangles Mesh - Cell Based Spacing Information

As mentioned in the test case section, a triangle mesh of a ramp was used, and CFD solutions were obtained of an air flow approaching the ramp at mach 2. The mesh and the visualization of the density solutions are shown in figure 6.8. Figure 6.9 depicts the mesh obtained from the extents belonging to the tensors that were included in the tensor file. The values used for the parameters P and the new adaptive function (see the equidistribution section of Chapter 3) were 2 and 0.00198, respectively. Finally, a close up of a region with tensors, plotted as ellipses, is shown in figure 6.10. The ellipsis were generated by VisIt using the components of the tensor matrices.

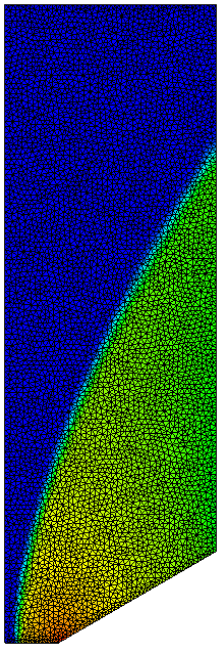

Figure 6.8 Density solutions on ramp mesh

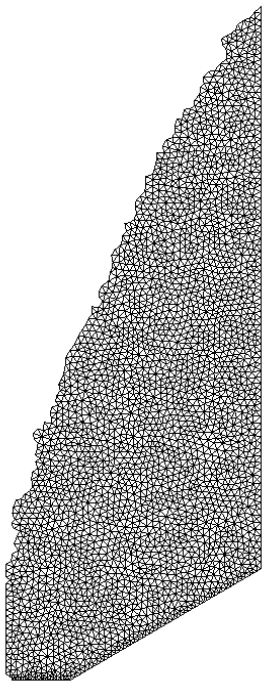

Figure 6.9 Cell extents mesh

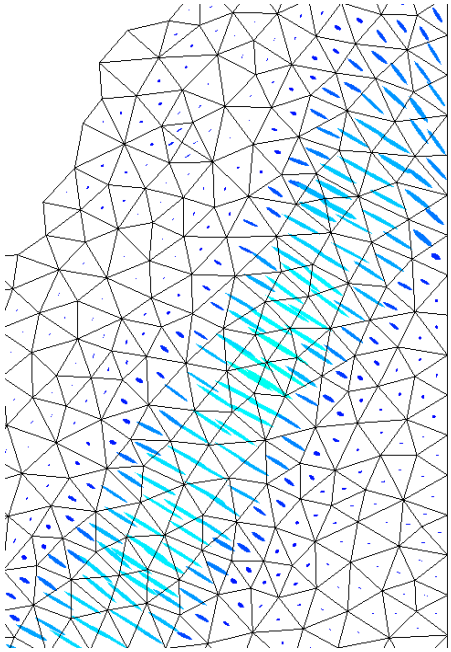

Figure 6.10 Close up of tensors plotted on cell extents mesh

Triangles Mesh - Node Based Spacing Information

Next, the mesh generated by the extents of spacing information stored at nodes is presented in figure 6.11. Figure 6.12 is a close up of the original mesh and the extent mesh overlap. Similarly as in the cell based calculations, the values used for the parameters P and the new adaptive function were 2 and 0.00198. Figure 6.13 shows the tensors, plotted as ellipses, at a region of the extents mesh. The ellipsis were generated by VisIt using the components of the tensor matrices.

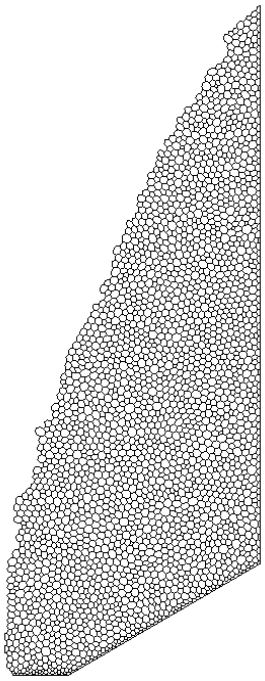

Figure 6.11 Dual extents mesh

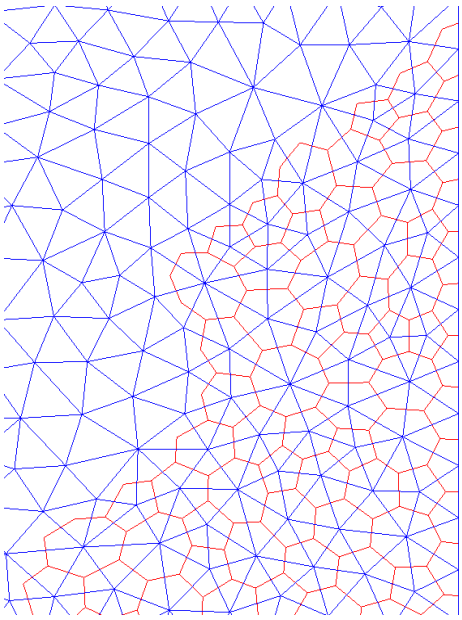

Figure 6.12 Close up of overlay of dual and original meshes

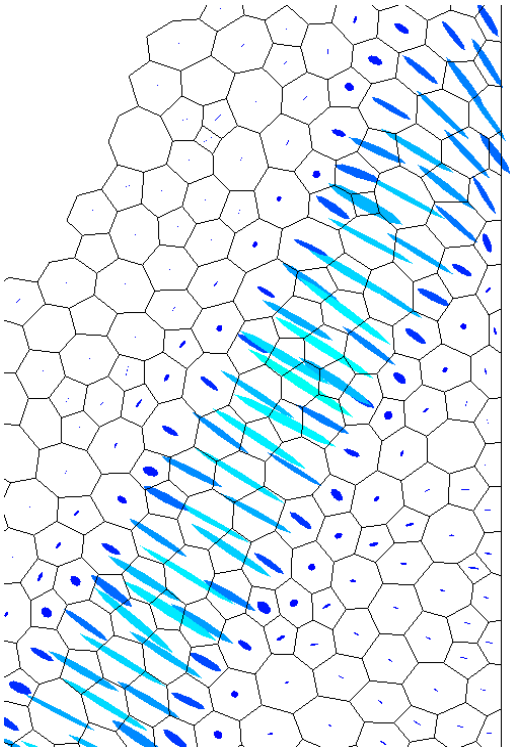

Figure 6.13 Close up of tensors plotted on dual mesh

Executing the Program With the `Optimize_Field` Function

As mentioned in Chapter 4, the development of the `Optimize_Field` function has not been completed yet due to a problem found and time constraints. However, using it at its current level of development has helped the visualization of what the extents merged into regions would look like. This was done by giving the extents tags of the regions they would belong. The `Optimize_Field` also has helped visualize the regions causing the difficulties.

Extents to Be Merged Into Regions

The next figures show some merged regions identified through tags. Figure 6.14 shows samples of extents merging into regions for both cell extents and centrodial dual extents. The number tags in the extents show the group of extents that would belong to one region.

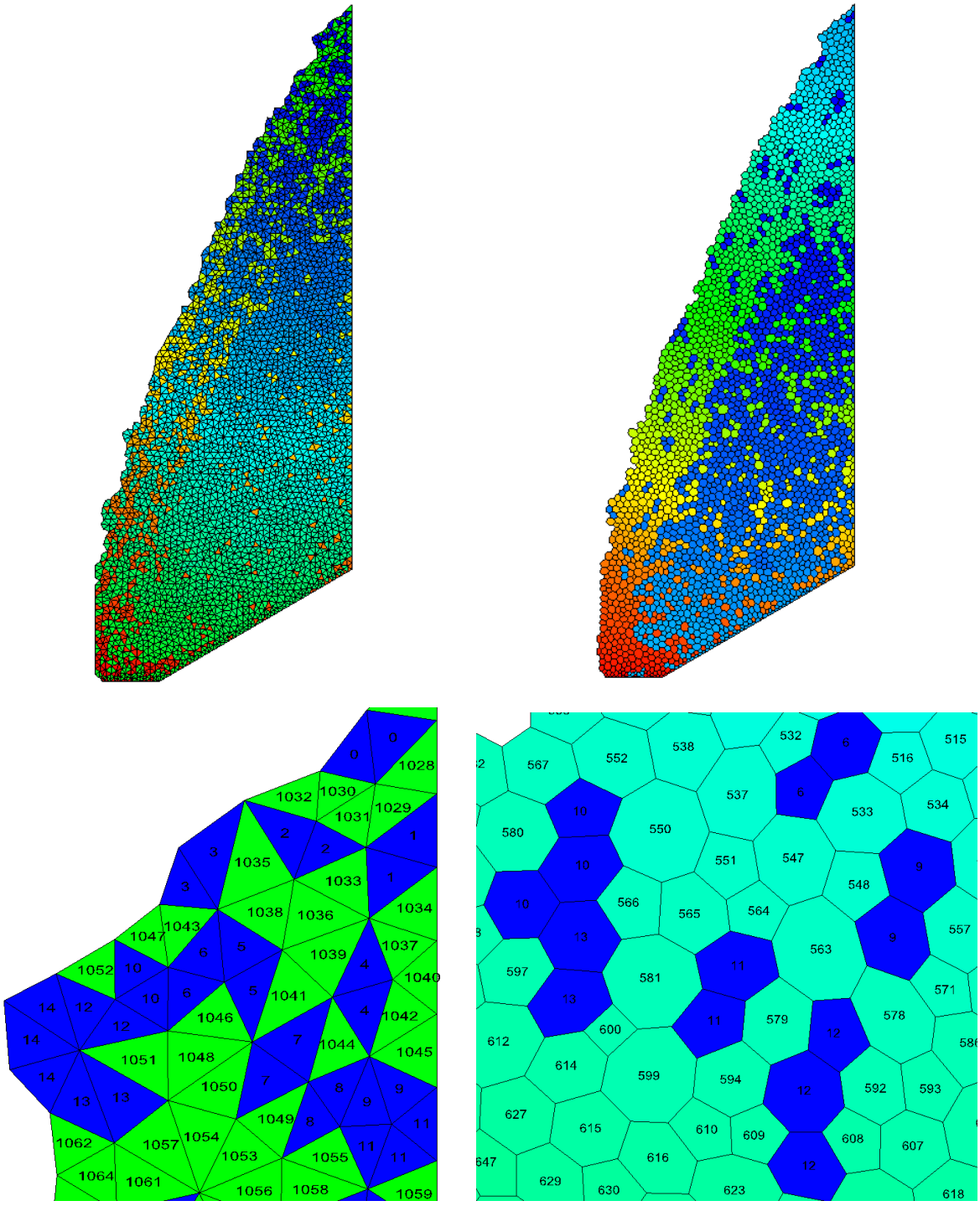

Figure 6.14 Merged extents

It is necessary to clarify that the value of the tags has no correspondence to the values of the solutions. The tagging was done in a way that the regions could be visually identified by taking advantage of the coloring scheme provided by the visualization software.

The Flaw

A problem was found when reordering the edges defining the region boundaries. There would be regions for which the edges could not be consecutively organized because the first edge in the region boundary could not be connected to the last one. After looking at plots of the extents, it was realized that there were multiple bounded regions enclosing others with a different tensor, as sketched in figure 6.15. This is the reason why the nodes of the inside boundary of the enclosing regions could not be attached to the nodes of the outside boundary through an edge. Figure 6.16 shows samples of multiple bounded regions enclosing others, for both cell extents and centroidal dual extents.

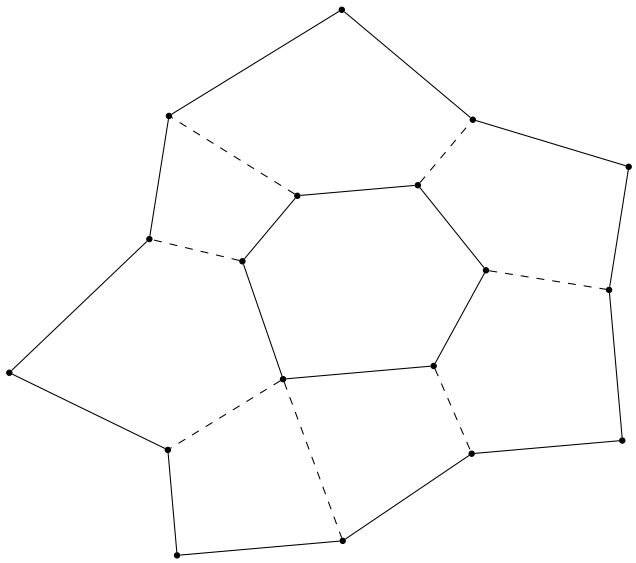

Figure 6.15 Sketch of multiply bounded region

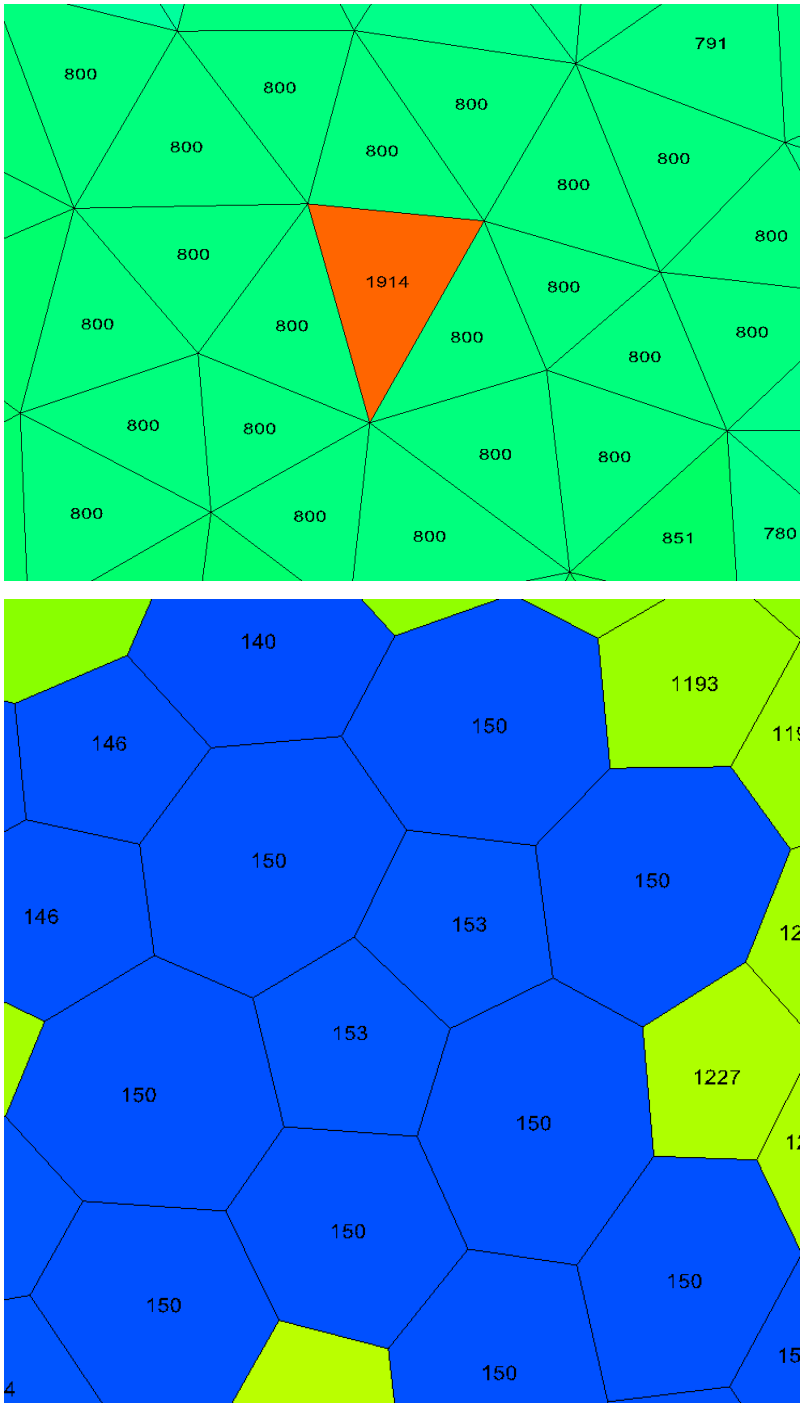

Figure 6.16 Multiply bounded regions

CHAPTER 7

THE SPACING LIBRARY

Definition

The Spacing Library is a library of functions developed for handling spacing fields. It makes use of the `Spacing_Obj and Quadtree_Storage` classes, and the member functions of those classes. This way, a small number of functions are available to the developer while all the operations are hidden. The motivation for developing this library was to minimize development of code devoted to handling files and manipulating spacing information. The tasks that can be performed by the library functions range from loading data from a tensor file to a quadtree, to retrieving spacing information, to computing metric lengths. A description of these functions are provided in the next section.

Library Functions

- `SF_Initialize`: It takes a string of characters that holds the name of the tensor input file. The function initializes a `Spacing_Obj` object, imports the data from the tensor file, initializes a `Quadtree_Storage` object, and populates the quadtree with the information in the input file.

- `SF_Finalize`: It takes no arguments. The function deletes the `Spacing_Obj` and `Quadtree_Storage` objects.

- `SF_Number_of_Entries`: It takes no arguments. Retrieves the number of entries in the `Spacing_Obj` object (same number of entries in the quadtree).

- `SF_Retrieve_Size`: It takes the coordinates of a point. Returns a scalar spacing value at the location of the point.

- `SF_Retrieve_Edge_Size`: It takes the coordinates of the points that define an edge. Returns a scalar spacing value for the entire edge.

- `SF_Retrieve_Tensor`: It takes the coordinates of a point. Retrieves a Riemann tensor containing spacing information at the location of the point.

- `SF_Retrieve_Edge_Tensor`: It takes the coordinates of the points that define an edge. Retrieves a Riemann tensor containing spacing information for the edge.

- `SF_Retrieve_Tensor_Item`: It takes the index of the requested item. The function retrieves all these data from the object with the requested index.

- `SF_Edge_Metric_Length`: It takes the coordinates of the points that define an edge, and the type of calculation for the metric length. Depending on the type, the function retrieves a metric length for the edge. If type is zero, the metric length is the largest obtained from the objects that hold pieces of the edge in it. If type is one, the metric length is a weighted average of the metric lengths obtained from the objects that hold pieces of the edge. The weighting function used is the portion of the edge that the objects contain.

- `SF_Compute_Metric_Length`: It takes the coordinates of the points that define an edge, and a Riemann tensor. The function computes an edge vector from the points, and retrieves the computed metric length resulting from such vector and the Riemann tensor.

- `SF_Decompose_Tensor`: It takes a Riemann tensor. The function decomposes the tensor into the rotation and scaling matrices.

- `SF_Compute_Riemannian_Metric`: It takes the principal direction vectors, and a value for the spacing in each direction. The function uses these arguments to calculate and return a Riemannian tensor.

The functions above, as defined in the library header file, is provided in appendix A.

Spacing Library Performance For Retrieving Information

A timing test was set to measure the performance of the spacing library for retrieving spacing information with respect to retrieving information using the "brute-force" method. The test consisted in requesting, from each method, spacing information from several point locations in a spacing field. The time used for each retrieval was measured and added for all the requests, and the total times were compared.

Retrieving Methods

The brute-force is the baseline method. It consisted in visiting each spacing entry's extent in an array of spacing objects to find the extents containing the point. Then each spacing value was compared and the minimum was returned. The first step for using this method was reading the spacing and extent information from a tensor file. Memory was allocated as necessary, a `Spacing_Obj` class was started, and the spacing field entries were stored in a `Spacing_Obj` array using the `Store_Tensor` function. Every time information was requested at a point, the `Point_In_Element` function from the `Spacing_Obj` class was used for testing if the requested location was contained by each visited extent. Once a extent containing the point was found, the `Retrieve_Tensor_Item` function from the `Spacing_Obj` class was used to retrieve the spacing information. Then, the tensor stored at the extent was decomposed using the `Decompose_Tensor` function, and the spacing values in each principal direction was obtained from the entries in the scaling matrix. Then, the tensor for the extent

with the minimum spacing value was returned. Once testing was over, the `Spacing_Obj` class was deleted and memory was freed.

The method of using the Spacing Library only required calling three library functions. The first one was the `SF_Initialize`. Then, every time information was requested at a location, the `SF_Retrieve_Tensor` function was used. Once the test was over, the `SF_Finalize` function was called. Throughout the process, the library made use of a quadtree for organizing and retrieving information, and made use of some `Spacing_Obj` and `Quadtree_Storage` class functions. However all these operations were hidden in the test.

Test Mesh and Spacing Field

Simple meshes and spacing fields were used for the test. The first mesh and the scalar functions are the same ones presented for the first case in the test cases section of Chapter 6. The number of nodes in the mesh is 2601. For convenience, the mesh and the plot of the scalar functions are presented below in figure 7.1.

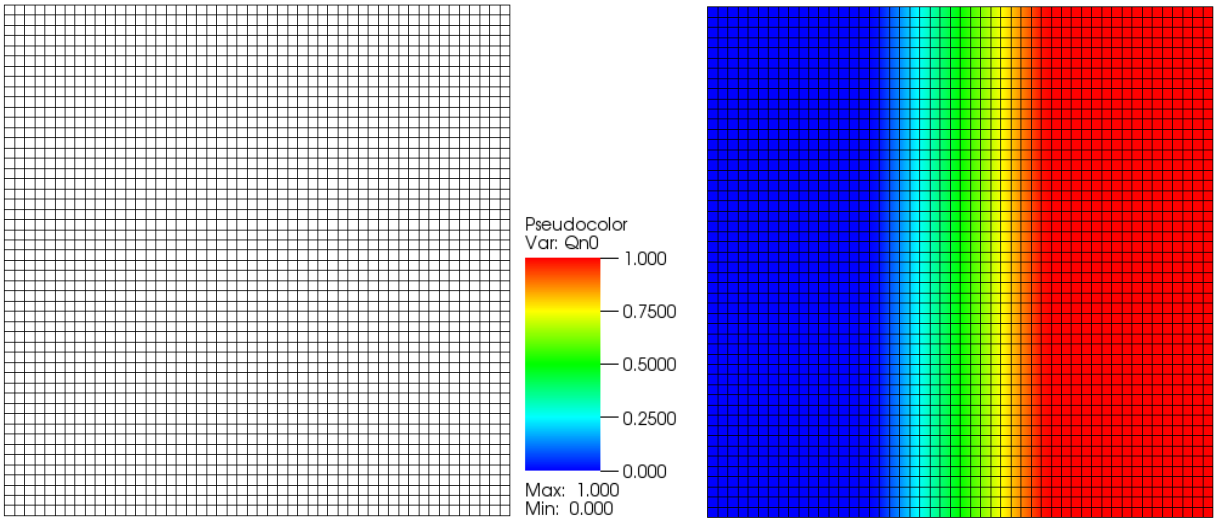

Figure 7.1 Square mesh and scalar function used for test case

The second mesh is a square mesh of triangles. The number of nodes in this mesh is 2998. The dimensions and the number of points at the boundaries of the triangle mesh are the same as in the square mesh. The scalar function for this mesh was also set up similarly as for the first one. Figure 7.2 shows the triangle mesh and a two plots of the scalar function.

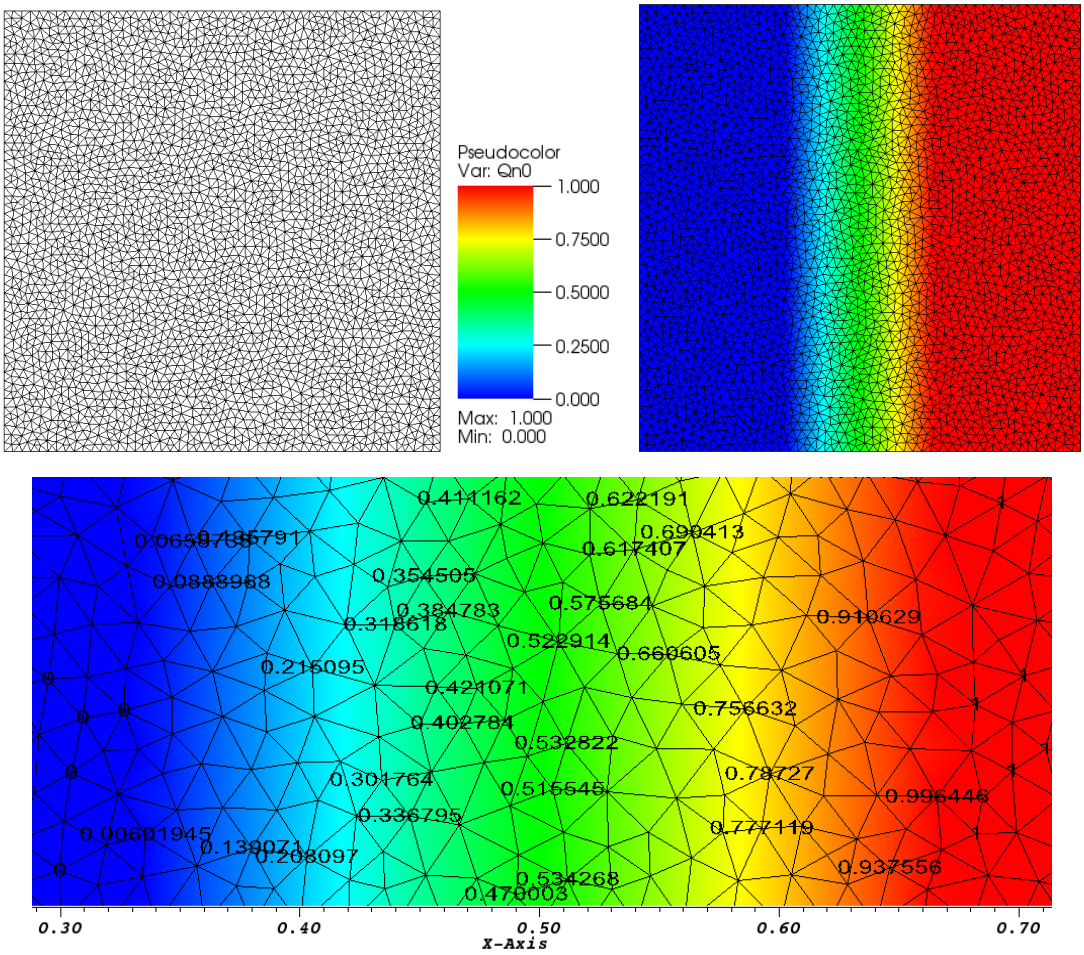

Figure 7.2 Triangle mesh and scalar function

The spacing information generated for both meshes were calculated at the nodes. For the square mesh the parameters used for P and $Af_{threshold}$ were the same ones used in the first test case presented in Chapter 6. In the case of the triangle mesh, the parameters P and $Af_{threshold}$ were 1 and 0.0129. Two spacing fields were generated for each case. One using

single squares as extents, and the second using centroidal duals as extents. The number of extents is the same as the number of nodes in the meshes. The fields for each mesh are depicted in figure 7.3. The single square extents mesh and the dual extents mesh of the square mesh are at the top, and the single square extents mesh and dual extents mesh of the triangle mesh are at the bottom.

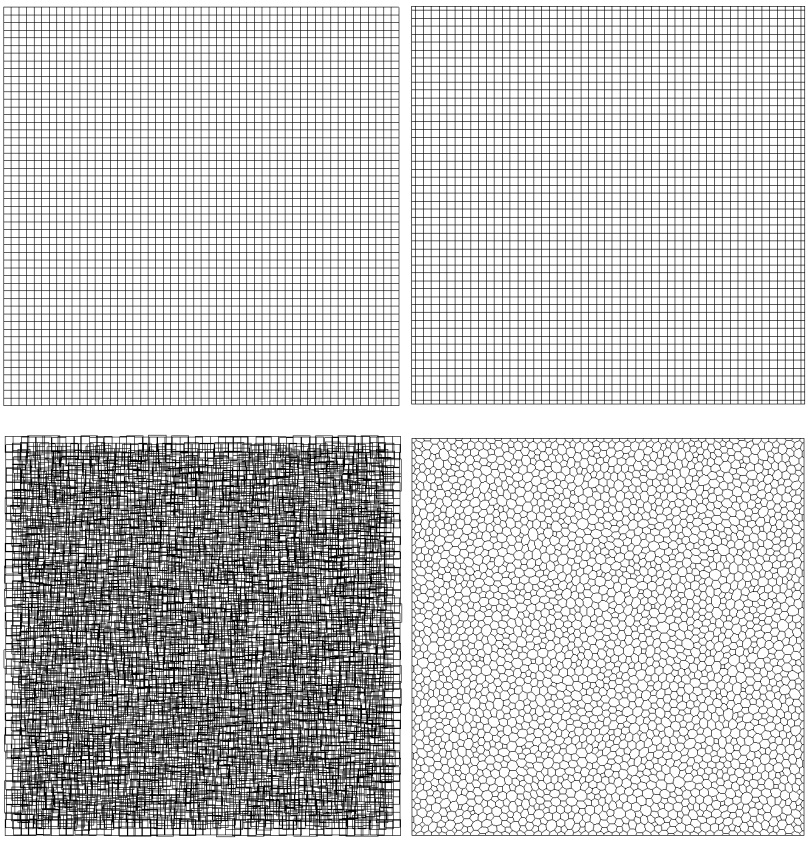

Figure 7.3 Single square extents and dual extents for each test case mesh

Point Locations of Requested Information

The point locations from where spacing information was requested were the points from the triangle mesh.

Results

The total time used to retrieve information at the requested locations are presented in table 7.1 for the square mesh, and table 7.2 for the triangle mesh. The tables show the time, in seconds, it took each method to retrieve information at all locations requested from both spacing fields for each mesh.

Total Time - Square Mesh		
Method	Square	Dual
Brute-Force	0.67831	5.32642
Spacing Library	0.04104	0.02283

Table 7.1 Comparison of timing each method using the square mesh

Total Time - Triangle Mesh		
Method	Square	Dual
Brute-Force	0.82233	8.87041
Spacing Library	0.06750	0.02816

Table 7.2 Comparison of timing each method using the triangle mesh

From comparing the times for the Spacing Library and the brute-force methods for the square mesh test case, it is observed that using the library is 16.5 times faster than using the brute force if information is retrieved from a spacing field with single square extents, and 233 times faster if the information is retrieved from a field with dual extents. In the case of the spacing field of the triangle mesh, using the library is 12.2 times faster than using the brute force if information is retrieved from a spacing field with single square extents, and 315 times faster if the information is retrieved from a dual extents field.

The higher performance of the Spacing Library over the brute-force method is accredited to the use of the quadtree. Using the quadtree is the main difference between both methods. Recalling the algorithm for retrieving information, the `Retrieve_List` function of the `Quadtree_Storage` class always follows the path of the most possible location of the

point where information is requested, as opposed to checking every single item in a list for every location.

Using the Spacing Library With an Optimized Field

Due to the simplicity of the square test mesh, it was possible to successfully use the `Optimize_Field` function when running the `create_spacing` program and saving information at the nodes with dual extents. The optimized spacing field is depicted by figure 7.4.

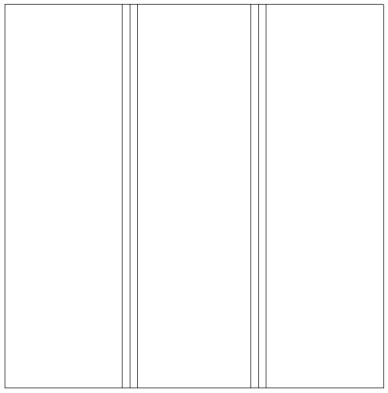

Figure 7.4 Optimized spacing field for the square mesh.

The total time it took retrieving information at the requested locations using the Spacing Library was 0.01127 seconds. It is observed that, for this particular test case, using the library with an optimized field resulted almost twice as fast as using the library with a not optimized field. However, more test cases should be used to arrive to a final conclusion on the efficiency of using optimized fields with respect to using not optimized fields.

CHAPTER 8

CONCLUSIONS

Spacing fields are a collection of items that specify desired spacings in different regions of the mesh. They are used in the mesh generation process and adaptive Winslow elliptic smoothing. The desired spacing information is used to specify edge sizes in all directions through scalars, or in two principal directions through Riemannian tensors. Defining spacing is most commonly done through Riemannian tensors.

Desired spacings are calculated using the concept of equidistribution and CFD solution gradients. The idea of equidistribution is to equally distribute a weighting function for all elements across a mesh using an equation (3.6) to calculate the value of an adaptive function that is common for all elements across the mesh. The gradients are obtained using a derivation of the theorems of Green and Gauss. They are calculated from CFD solutions obtained from previous simulations on the mesh. The gradients are also used to derive the principal directions.

The regions on which the desired spacings apply are delimited by the extents. The extents are defined depending on elements of the mesh for which the spacing information is calculated. If spacing information is obtained for mesh cells, the extents are the cells themselves. If spacing is obtained for mesh nodes, the extents can be defined through single boxes or through centroidal duals. The use of duals is recommended in order to avoid extent overlapping.

Quadtrees are used for organization of spacing data. The purpose is to speed up the process of retrieving information. Using quadtrees, spacing information can be searched

based on the location where it applies in the mesh, and not the location of its spacing item in an array.

The contributions in the present work are the following:

1. The creation of the `create_spacing` program: The program takes mesh and CFD solution information and defines a spacing field by calculating desired spacing for nodes or cells. The program exports the information to a tensor file.

2. The development of the `Optimize_Function`: The motivation was to improve the efficiency of the process of retrieving information. The idea was to reduce the number of entries in the spacing field by merging extents with similar spacing information. Although still in development, it has helped visualize what extents merged into regions would look like. Also, it has allowed finding the problem of the algorithm currently used to merge the extents.

3. The creation of the Spacing Library: The Spacing Library was developed to manage input files, and to organize, retrieve, and manipulate spacing information so that others do not have to. It has the potential of saving mesh researchers code development time spent on creating routines to perform the tasks mentioned above. The functions in the Spacing Library have already shown to be useful in the research work of Druyor and O'Connell, master degree candidates at The University of Tennessee at Chattanooga [15] [16].

CHAPTER 9
FUTURE WORK

More Testing Cases

The create `create_spacing` program and the Spacing Library have been developed using the three meshes mentioned in the test case section of Chapter 6. More testing is necessary with meshes of other geometries, such as airfoils.

Fixing Region Boundaries

The last subsection of Chapter 4 discusses a problem found with the generation of regions by merging extents. Regions that multiply connected have been detected. These regions have two sets of boundary edges (maybe even more) which need to be joint in some way. A proposed idea is to preserve one or more edges in the edge deletion process (see "The Optimize Field Function" section of Chapter 4). Another option is to pick a vertex in the outer boundary, search for the closest vertex in the inner boundary, and create a new edge to join both boundaries. These and other ideas need to be explored and implemented to fix the issue. Then, more testings, similar to the one used in the last section of Chapter 7, need to be performed to analyze the gains in efficiency by merging extents.

Designing Algorithms to Split Extents and Fit Them on Subdivisions' Extents

The last subsection of Chapter 5 discusses a source for inefficiency in the process of retrieving information from the quadtree. It was mentioned that every time the spacing information was requested at a certain location of the mesh, a relatively large set of items

that could possibly contain such location was retrieved. Then, every item extent was tested to see if it contained such location. It was also said that, because testing extents for location of points was computationally demanding, the process of retrieving information this way was slow. In order to alleviate this issue, it is necessary to change the way information is stored and arranged in the quadtree. A proposed idea is to split the extents in such a way that the shapes and sizes of the new extents fit exactly the extents of relatively fine subdivided squares at some high layer of the tree. By doing this, the tree becomes flatter and all entries are stored at some finer level. This would allow the information retrieving algorithm to use the quadtree to arrive precisely to the extent that contains the location of the information requested, or having to test very few extents before getting to the needed one. This and other ideas need to be studied and the solution needs to be implemented.

Extrapolating Work to 3D

The knowledge and experience gained from this research work are mostly useful for later implementation of the `create_spacing` program and the Spacing Library in 3D. Therefore, the next step is to implement these algorithms for three dimensional meshes.

REFERENCES

[1] JR, D. S. K., "Grid Generation Lecture," Introduction to grid generation. 1

[2] "www.pointwise.com", "Pointwise," . 1, 3

[3] Anderson, D. A., "Equidistribution Schemes, Poisson Generators, and Adaptive Grids," *Applied Mathematics and Computation*, Vol. 24, 1987, pp. 211–227. 3, 7

[4] "www.pointwise.com/gridgen", "Gridgen," . 3

[5] Sahasrabudhe, M., *Unstructured Mesh Generation and Manipulation Based on Elliptic Smoothing and Optimization*, Ph.D. thesis, The University of Tennessee at Chattanooga, August 2008. 3

[6] Masters, J. S., *Winslow Elliptic Smoothing Equations Extended to Apply to General Regions of an Unstructured Mesh*, Ph.D. thesis, The University of Tennessee at Chattanooga, December 2010. 4

[7] JR., S. K. and Sahasrabudhe, M., "Unstructured Adaptive Elliptic Smoothing," No. 0559, AIAA, American Institute of Aeronautics and Astronautics, Inc., January 2007. 4, 5, 8

[8] JR, D. S. K., "Adaptive and Dynamic Mesh Lecture," Winslow-Laplace Smoothing. 6

[9] Kreyszig, E., *Advanced Engineering Mathematics*, John Wiley and Sons, Inc., 9th ed., 2006. 6

[10] JR., S. K. and Wooden, P., "CFD Modeling of F-35 Using Hybrid Unstructured Meshes," No. 3662, AIAA, American Institute of Aeronautics and Astronautics, Inc., June 2009. 12

[11] Kamfonas, M. J., "Recursirve Hierarchies: The Relational Taboo!" *The Relational Journal*, October/November 1992. 27

[12] "www.vtk.org", "Visualization Toolkit," . 45, 46

[13] "wci.llnl.gov/codes/visit", "VisIt," . 46, 49

[14] Erwin, J. T., "2D Euler Solver," Euler Solver for 2D Meshes. 49

[15] JR., C. D., *An Adaptive Hybrid Mesh Generation for Complex Geometries*, Master's thesis, The University of Tennessee at Chattanooga, August 2011. 72

[16] O'Connell, M., *Comparison of Two Methods for Two Dimensional Unstructured Mesh Adaptation with Elliptic Smoothing*, Master's thesis, The University of Tennessee at Chattanooga, August 2011. 72

APPENDIX A

SPACING LIBRARY FUNCTIONS AS DEFINED IN SOURCE CODE

- `int SF_Initialize(char fname[]);`
- `int SF_Finalize();`
- `void SF_Brute_Search_Timing(int nNodes,double * x,double * y);`
- `int SF_Number_of_Entries();`
- `double SF_Retrieve_Size(double pt[SD]);`
- `double SF_Retrieve_Edge_Size(double p1[SD],double p2[SD]);`
- `int SF_Retrieve_Tensor(double pt[SD],double rmt[SD][SD]);`
- `int SF_Retrieve_Edge_Tensor(double p1[SD],double p2[SD],`
  `double rmt[SD][SD]);`
- `int SF_Retrieve_Tensor_Item(int n,double rmt[SD][SD],int &nvrt,`
  `double vert[][SD]);`
- `double SF_Edge_Metric_Length(double p1[SD],double p2[SD], int type);`
- `double SF_Compute_Metric_Length(double p1[SD],double p2[SD],`
  `double rmt[SD][SD]);`
- `void SF_Decompose_Tensor(double RT[SD][SD],double left[SD][SD],`
  `double right[SD][SD],double lam[SD]);`
- `void SF_Compute_Riemannian_Metric(double e1[SD],double e2[SD],`
  `double h1,double h2,double RT[SD][SD]);`

VITA

Max David Collao was born in Lima, Peru on September 30th, 1984. He is the son of Maximo and Gloria Collao, and elder brother to Joel, Jairo, and Elisabet Collao. He attended high school to CNMx. San Felipe, and graduated in December of 2000. David came to the United States in the Spring of 2004. In May of 2009 he earned a Bachelor's degree in Mechanical Engineering from Lipscomb University, in Nashville, TN.